ABSTRACT

Evaluating Impulse C and Multiple Parallelism Partitions for a Low-Cost Reconfigurable Computing System

Carmen C. Li Shen, M.S. Electrical and Computer Engineering

Advisor: Russell W. Duren, Ph.D.

Impulse C is a C-to-HDL compiler from Impulse Accelerated Technology that facilitates the introduction of software programmers, mathematicians, and scientists, into the realm of FPGA-based algorithm development for high-speed numerical computation. This thesis evaluates the Impulse C programming language and explores differing levels of parallelism across multiple, homogeneous, FPGA development platforms using the Aurora serial communication scheme. Impulse C and Xilinx IP cores are employed in the numerical computation of a neural network consisting of 27 inputs and 1200 outputs. The artificial neural network is capable of emulating an underwater acoustic environment and has been used to determine characteristic parameters of reflections from the ocean floor. Timing, logic utilization and ease-of-use are metrics used to evaluate Impulse C in the automatic generation of VHDL code for the network test application. Implementations with parallelism at the system level and at the intermediate (loop) level are explored as part of this study.

Evaluating Impulse C and Multiple Parallelism Partitions
for a Low-Cost Reconfigurable Computing System

by

Carmen C. Li Shen, B.S.

A Thesis

Approved by the Department of Electrical and Computer Engineering

____
Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

____
Russell W. Duren, Ph.D.,Chairperson

____
Steven R. Eisenbarth, Ph.D.

____
David B. Sturgill, Ph.D.

Accepted by the Graduate School
December 2008

____
J. Larry Lyon, Ph.D., Dean

`

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ANN - Artificial Neural Networks

ANSI - American National Standards Institute

BRAM - Block Random Access Memory

CPU - Central Processor Unit

CSP - Communicating Sequential Processes

DDR SDRAM - Double Data Rate Synchronous Dynamic Random Access Memory

FIFO - First-In First-Out

FPGA - Field Programmable Gate Array

GCC- GNU Compiler Collection

HDL - Hardware Description Language

HL- High Latency

HLL - High Level Language

IAT - Impulse Accelerated Technology

IDE - Integrated Development Environment

IEEE - Institute of Electrical and Electronics Engineers

ISA - Instruction Set Architecture

LL - Low Latency

MAC - Multiply and Accumulate

MGT - Multi-gigabit Transceiver

OPB - On-chip Peripheral Bus

PLB - Processor Local Bus

SATA - Serial Advanced Technology Attachment	
SMD - Stage Master Debugger	
SME - Stage Master Explorer	
VHDL - VHSIC Hardware Description Language	
VHSIC - Very-High-Speed Integrated Circuits	
XUP - Xilinx University Program	

ACKNOWLEDGMENTS

I am grateful to God's for His daily blessings and the strength He gives me.

I would like to thank Dr. Duren for his guidance and encouragement throughout the development of this thesis work. I would like to thank Dr. Eisenbarth for his help in revising this thesis and providing me with insights in some problems. I would like to thank Dr. Sturgill for providing me with some C coding tips and comments to improve my thesis readability. I would also like to thank Impulse C staff for their response to my questions on CoDeveloper tools. And lastly, thanks to Anthony Chu and friends for the encouragement and support I received throughout the course of this work.

DEDICATION

To my parents
for believing in me and their sacrifice for me to have an education

CHAPTER ONE

Introduction

Neural networks are mathematical models trained to make decisions based on patterns imitating biological neurons in behavior. These models have been used in diverse applications, including remote sensing and identification, adaptive control, optimization, signal filtering, complex mapping, to mention a few. One particular area of interest is exploring the feasibility of implementing a neural network-based application for sonar acoustic model emulation. The sonar acoustic model is a very reliable yet complicated algorithm requiring a large amount of computation, which makes it difficult to produce real-time results. For this reason, a neural network model was created that emulates the behavior of an acoustic system which greatly reduces computation levels. Even with the significant improvement generated by the neural network model, the computation level still prevents a successful real-time implementation when using traditional processor-based computing environments [1]. Neural networks are characterized by the presence of highly parallelizable computations, which require hardware architectures with the capability to exploit this parallelism. Therefore, as an alternative solution, FPGA-based hardware assist was considered for the neural network implementation. These reprogrammable devices (FPGAs) provide great flexibility and potential for speedup inherent in hardware oriented parallelism. In previous work, an acoustic model emulator, which consisted of multiple feed-forward neural network passes, was implemented on a FPGA-based platform. The results of the study by Reynolds, et al. [1] indicate the feasibility of a real-time implementation after extensive hand-coding optimization techniques were applied to a VHDL model. The outcome of the implementation was encouraging, but the level of hardware description language knowledge needed to develop the optimized algorithm represents an obstacle for those that do not specialize in the hardware design arena. Designers, scientists and general engineers that could benefit from the computational advantages and speed of FPGAs are reluctant to utilize them for this reason. Recent development of C-to-HDL compiler technology has eased the gap between software developer's experience-level and the expertise needed to produce hardware implemented algorithms. Impulse C is a C language-based compiler that facilitates the co-design of mixed HW/SW applications by automatically generating HDL code in the form of VHDL.

Since neural network implementations in FPGAs, with low-level language optimization, can provide real-time results, our aim was to assess the feasibility of implementing (on FPGA-based platforms) a neural network application designed in a high-level language environment and to obtain comparable real-time results. For this purpose, Impulse C was employed for the hardware implementation of a neural network application.

The focus of this thesis is to evaluate the Impulse C programming language and explore differing levels of parallelism across multiple, homogeneous, FPGA development platforms using the Aurora serial communication scheme. The particular application studied is a neural network algorithm for a sonar acoustic problem. Serial and parallel versions of the algorithm for the neural network computations are created and simulated with Impulse C. The VHDL code automatically generated is then synthesized using platform-specific tools and the resulting bit files are used to reprogram the FPGA. The floating-point based implementation of the neural network does not fit into a single XUP platform, requiring multiple XUP boards to provide sufficient logic for the complete application. These platforms can communicate with each other through various serial and parallel protocols, for our case they are connected via SATA ports using the Aurora communication scheme. The objective is to partition the computations into hardware leaving the communication at the software level.

A secondary emphasis of this research is the comparison of the timing results of the neural network computations achieved by the different neural network implementations. The results from this current research will be compared to previous (fixed-point based) work that emphasized hardware/software co-design using the SRC-Carte and VHDL languages on SRC-6e or multiple XUP Virtex II Pro hardware development platforms. Therefore, some basic understanding of the platform and the languages is necessary. Having this in mind, chapter two provides some background information on reconfigurable computing and a description of the different FPGA computing platforms. Chapter three will serve as a description and comparison of the Impulse C and VHDL program development environments. Impulse C and VHDL are contrasted with high-level languages and the main characteristics of Impulse C are explored. Chapter four describes the neural network application being implemented and related research on the realization and use of neural networks. Chapter five explains the different Impulse C design and implementation techniques and the hardware/software partition algorithms tested. Timing results obtained via the Impulse C cycle-accurate debugger and the OPB timer are then discussed and compared to previous work results in chapter six. Chapter seven summarizes the object lessons learned and possible future research implications.

CHAPTER TWO

Background Information

Reconfigurable Computing

Although reconfigurable computing, as a concept, dates back to the 60's, it was not until the advent of the Field Programmable Gate Array (FPGA) in the 80's, that reconfigurable computing came of age. Advancements in device technology and manufacturing techniques have created larger and less expensive FPGAs, which have spurred their use where fast computation or complex applications require a significant degree of flexibility.

Reconfigurable computing is the process of re-tasking or reprogramming the logic blocks that compose the internal organization of a computing system; many of which are populated with FPGA components. Reconfiguration increases the inherent functionality in these computing systems. In addition, because FPGAs are prefabricated, their cost is greatly reduced over custom, application-specific designs. Although FPGA-based designs may provide sub-optimal computation power over application-specific designs they are typically more flexible and ultimately cheaper because they are reprogrammable. Even though FPGAs may not achieve the clock frequency of von Neumann processors (compare 100 MHz to 3 GHz), they can still provide significant speedup when the resources of the FPGA are optimally used and an appropriate programming methodology is applied. In comparison to the software programs executing on traditional von Neumann architecture computing machines, FPGAs provide flexibility and occasionally greater efficiency because the FPGA's internal structures are actually being modified to change its functionally. Figure 1 shows the basic internal structure in a FPGA, which may possess one or more microprocessors, random-access memory, input/output devices and a large block of reconfigurable logic.

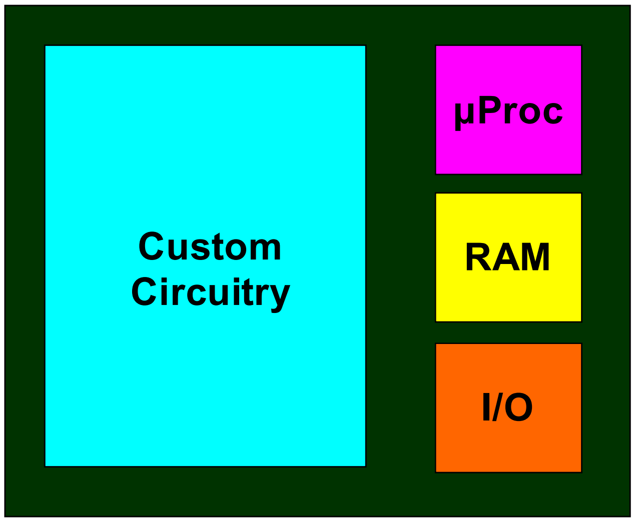


Figure 1: Components of a Typical FPGA

In the early stages of FPGA application development, designers hand picked [logic resources] gates and interconnecting circuitry. As the logic cell density of these devices increased, so did their complexity, calling for the use of Hardware Description Languages (HDLs) and Electronic Design Automation (EDA) applications capable of synthesizing the hardware description code and creating a physical design in terms of FPGA's resources. Advances in technology brought the FGPA onto Moore's Law curve, i.e. device density doubling every 18 months, but leaving application development on a decidedly non-Moore path.

Application development in the FPGAs field typically starts with the design's functionally being captured by a hardware description language such as Verilog or VHDL (VSHIC Hardware Description Language). However, the majority of software developers are accustomed to application development in the High Level Languages (HLLs), particularly ANSI C, C++ and Java. Software developers also suffer from limited experience with the details and intricacies of hardware development thereby limiting their ability to develop applications for FGPAs. Similar experience constraints also discourage other scientists and mathematicians from taking advantage of the FPGA solution space [2, 3].

HDLs are effective at describing hardware functionality, but a level of hardware expertise is required to utilize HDL effectively. To leverage the skills of software developers and reduce the impact of the hardware learning curve, EDA developers have produced tools suites that utilize ANSI C, C++, or Java-based programming languages such as Impulse C [4], System C [5], Join Java [6], among others. These programming languages make the translation from a high-level language to a low-level HDL essentially a black box for the programmer. These tool suits provide development and debugging capabilities that are very similar to software development environments. It is important to note that intimate knowledge of the target hardware on the developer's part can lead to highly efficient applications, however, the higher efficiency comes at the expense of increased development time and the additional expertise required.

Computing Systems

In Patel, et al. [7] reconfigurable architectures are classified as Class 1 Machines, Class 2 Machines and Class 3 Machines. Until recently, the trend in computer systems design has been to increase the system clock rate and exploit parallelism with multiple von Neumann processors (Class 1), but it has reached the point that doing so produces diminishing returns [8]. This bottle-neck has spurred the exploration of new computing architectures; one of which combines the serial processor and a FPGA (Class 2). The FPGA in this case was used as an external component to help speedup complex hardware calculations. Now, because reconfigurable devices are more accessible, FPGAs are used as standalone products; many of which include one or more embedded microprocessors (Class 3). The Patel article explains further the pros and cons of each category with examples and explanations of their different uses. Our study examines a “Class 3 Machine,” employing a combination of FPGA platforms, each with two on-chip processors. The neural network application will be implemented in this system and its performance will be contrasted with the outcome obtained in implementations on the SRC-6e reconfigurable computer. An understanding of the hardware resources available in each system puts into better perspective the resource utilization and the hardware impact in the execution of an algorithm. These two platforms are described in more detail in the next subsection.

SRC-6e

The SRC-6e[1] reconfigurable computer consists of two reconfigurable Xilinx XC2V6000 FPGAs that are programmed with a SRC system-specific language, two Pentium 3 microprocessors running at one hundred megahertz, and six 4 megabyte memory blocks. Each FPGA contains approximately six million logic gates, 144 eighteen-bit multiplier blocks and 144 eighteen-kilobit random-access memory blocks. Off-chip communication is done via three 64-bit bidirectional ports [1]. A diagram of the SRC-6e hardware architecture is shown in Figure 2. The SRC-6e system is a two-board

[1] Newer versions of the SRC-6e utilize faster Pentium microprocessors and faster interconnecting buses. They also come with a higher price tag.

unit, the first is the microprocessor board and the second contains all the reconfigurable resources of the system.

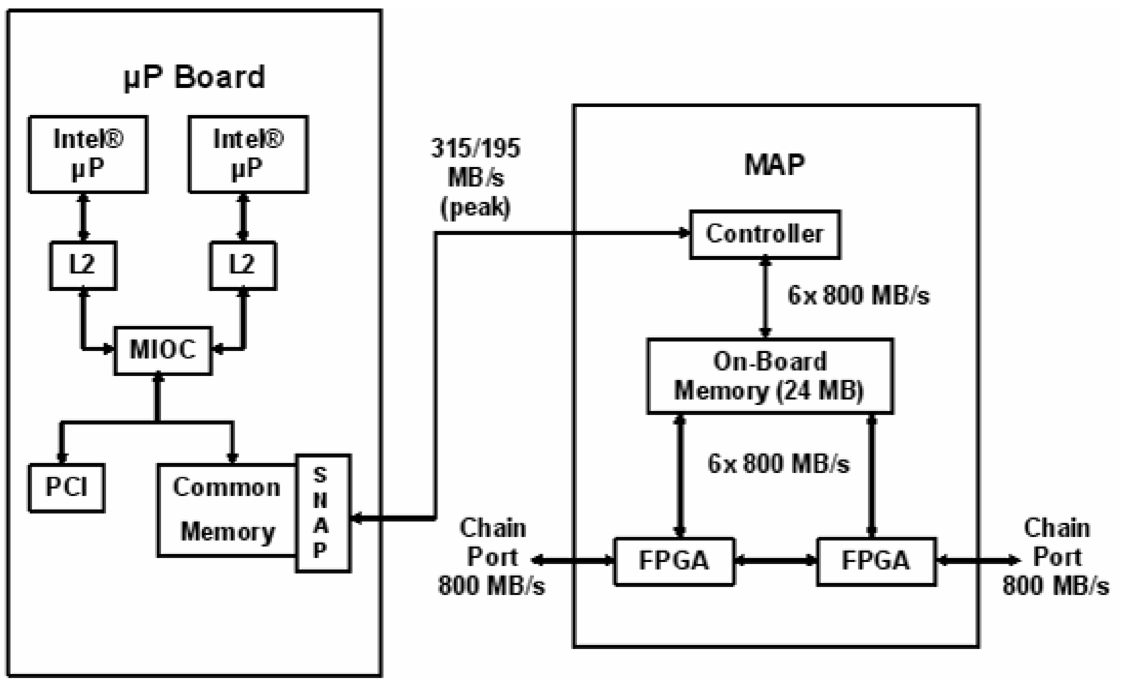


Figure 2: SRC-6e Hardware Architecture [1]

The SRC-6e system is expensive since it is targeted toward their proprietary computing platforms (SRC systems). The approximate cost for the prototype hardware and development system is $300,000. It has a C to VHDL compiler, Carte C, which is characterized for its ease-of-use, but the high price makes it a less attractive option. SRC-6e also incorporates user HDL code and IP cores if necessary.

XUP Virtex II Pro

The XUP Virtex II Pro is the platform board used for this evaluation. More precisely, a group of multiple XUP boards are used as the target platform with the aim to migrate the application to the Baylor University Reconfigurable Computing Cluster. This cluster is composed of sixteen XUP boards each of which is booted with a QNX OS image and maintains communications with all boards via an Ethernet stack and port. A QNX OS system running on an X-86 system provides "master" access to each XUP board in the cluster. The physical connection between boards is via an Ethernet port connected to a high-speed switch/router. Currently, a 16-node XUP system is in the final stages of testing.

The individual features of a XUP board is the following: one XC2VP30 FGPA with roughly three million logic gates, 136 eighteen-bit multipliers and 136 block RAMs, two hard core processors (IBM PowerPCs), and 256 MB of DDR SDRAM (for our implementation, but with a capability of up to two gigabytes of DDR SDRAM). The board is also populated with multi-gigabit transceivers, three of which are employed as SATA interface. It also contains multiple ports: JTAG, USB, RS232, PS2 for keyboard, and mouse, Ethernet and video and has capability for serial communication and audio interface. All the component and features of the XUP Virtex II Pro board are shown in Figure 3.

Synthesis and mapping to the XUP board requires a Xilinx synthesizer and other tools proprietary to this vendor. Tools such as Core Generator from Xilinx facilitate the implementation of cores for floating-point hardware, memories and FIFOs, serial communication protocols, as well as other intellectual property (IP) code as an aid for the system developer. These cores are customizable and have been optimized to obtain the best performance and board resource utilization [9].

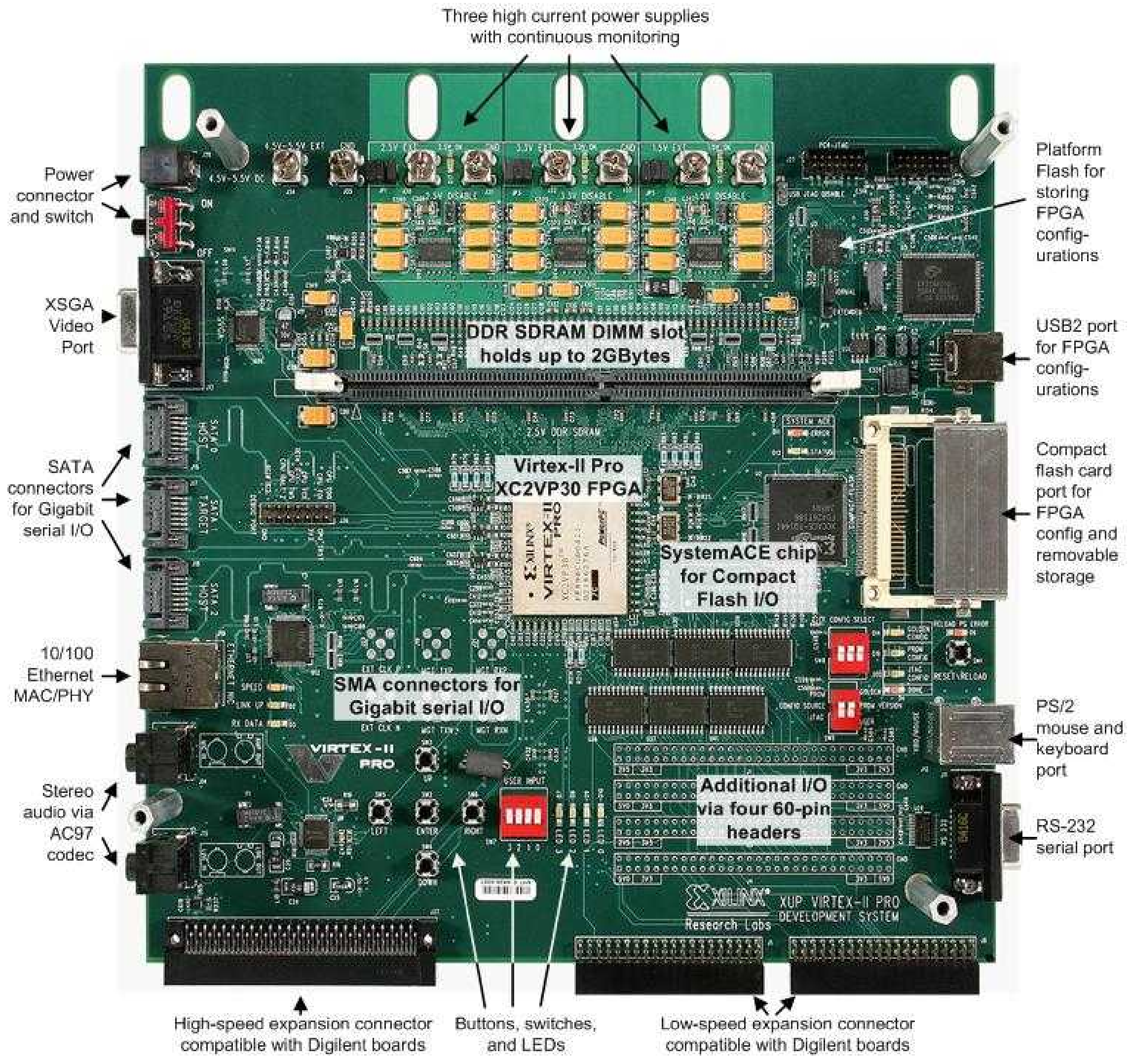

Three high current power supplies with continuous monitoring
Power connector and switch
Platform Flash for storing FPGA config-urations
XSGA Video Port
DDR SDRAM DIMM slot holds up to 2GBytes
USB2 port for FPGA config-urations
SATA connectors for Gigabit serial I/O
Virtex-II Pro XC2VP30 FPGA
Compact flash card port for FPGA config and removable storage
SystemACE chip for Compact Flash I/O
10/100 Ethernet MAC/PHY
SMA connectors for Gigabit serial I/O
PS/2 mouse and keyboard port
Stereo audio via AC97 codec
Additional I/O via four 60-pin headers
RS-232 serial port
High-speed expansion connector compatible with Digilent boards
Buttons, switches, and LEDs
Low-speed expansion connector compatible with Digilent boards

Figure 3: XUP Virtex II Board [9]

To communicate between boards the Aurora IP is employed. Aurora is a lightweight protocol for MGT (Multi-Gigabit Transceivers) links. In Figure 4, each lane used to communicate between Aurora interfaces represents a high-speed serial connection between MGTs. A group of these connections form an Aurora channel through which data can be sent. When the Aurora channel is not transmitting actual data, idle characters are transmitted to maintain channel timing, similar to the XAUI protocol [10].

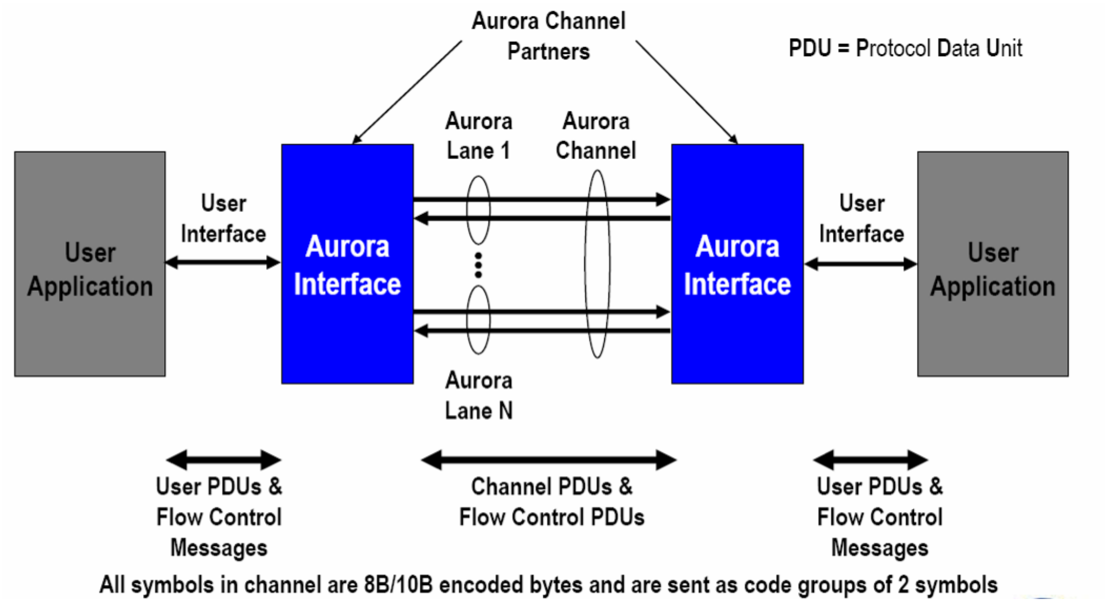

Aurora Channel Partners
PDU = Protocol Data Unit
Aurora Lane 1
Aurora Channel
User Application
User Interface
Aurora Interface
Aurora Interface
User Interface
User Application
Aurora Lane N
User PDUs & Flow Control Messages
Channel PDUs & Flow Control PDUs
User PDUs & Flow Control Messages
All symbols in channel are 8B/10B encoded bytes and are sent as code groups of 2 symbols

Figure 4: Aurora Communication [10]

Aurora provides two modes for communication: streaming and framing. Streaming is more or less like a pipe where the bare data is send by the transmitter to the receiver. It can also be considered to be one continuous frame. The streaming mode is easy to implement, but the user does not have much control. Framing on the other hand can be combined with flow control and data is sent in fixed length frames.

CHAPTER THREE

Programming Languages

Hardware Description Languages (HDLs)

A hardware description language provides a formal expression for the hardware (physical) component of an electronic system. HDLs, in contrast to high-level languages, indicate explicitly the timing behavior and concurrent connectivity between the logic blocks in a device. However, this form of description represents a lower degree of abstraction in comparison to the typical software languages. The two most representative HDLs are Verilog and VHDL [11]. VHDL will be explored further, but additional information about Verilog can be found at [12].

Very-High-Speed Integrated Circuits Hardware HDL (VHSIC HDL) was created as a request from the US Department of Defense to serve as a mean for documentation and reference of the ASICs' structure and functionality found in many of the devices the department acquired [11]. VHDL is useful for extracting the parallelism inherent in FPGAs. However, experience with the language and hardware is necessary due to its low-level abstraction. Another benefit of VHDL that attracts hardware developers is its standardization. It is applicable as a general-purpose language that allows the user to target a wide range of hardware configurations unlike some of the C-to-HDL compilers that are developed for a specific system. A big disadvantage of programming with VHDL is the time required to create test benches and obtain simulation results.

High Level Languages (HLLs)

High level languages are programming languages with higher levels of abstraction, portability (in most cases) and ease-of-use than their hardware description languages counterparts. They do not, however, guarantee as optimal a solution as HDLs. A key distinction between HLL and HDL is the lack of timing in HLLs. A HDL programmer using lower-level language constructs has more control over algorithmic development because timing constraints are taken into consideration. The lower-level languages offer direct manipulation and mapping of the hardware resources. On the other hand, someone programming with higher-level languages can focus on the algorithm without stressing on timing and hardware architecture. The tradeoff in the design of a product is between the performance (many times with metrics of speedup and logic area) and productivity (designer expertise and time of development).

HLLs facilitate mathematical computations with a range of data types, like the summation of an integer and a floating point type using implicit type casting. Casting operations are not so readily done with strongly typed HDLs. Another incentive for HLL usage is their fast prototyping capabilities which frees up time for the designer to explore different algorithms and techniques.

Until recently, floating-point implementations in FPGA hardware have not been practical due to the large consumption of chip resources. For this reason, fixed-point and integer representations dominate FPGA-based application. The most significant drawback of these representations is their inability to provide a large dynamic range for applications such as pattern recognition, robotics and target acquisition applications, which are traditionally floating-point-based, but FPGA resource constraints are such that fixed-point implementations have been the only practical engineering solution. Fixed-point is a very efficient solution; however, for certain algorithms, the precision may not be sufficient. As device density increases in advanced FPGA devices (e.g. Virtex 5, which provides 25 bit by 18 bit multipliers), floating-point data representation and operations are becoming easier to implement [13].

Commonly used HLL's include Ada, Prolog, C, C++ and Java. Of these languages, the majority of programmers are familiar with ANSI C, which is the reason behind the multiple attempts to create efficient compilers that will translate code written in C to a HDL representation that will target hardware devices.

C-to-Register Transfer Level (RTL) Overview

FPGAs and their in-circuit reconfiguration capabilities have stimulated interest in reconfigurable computing due to the potential for significant performance improvements in many applications, but the tools used to program and analyze these devices also play an important role. The development and "maturity" of automated tools, first proprietary EDA and now C-to-RTL or C-to-HDL compilers, have encouraged a more general audience, not just hardware developers, to move towards application development using FPGA-based hardware. EDA tools have been around for a long time, but C-to-RTL compilers are a new development.

The concept of parallel C can be counter-intuitive since ANSI C is an intrinsically sequential programming language that does not readily support parallelism. It is necessary to expand the coverage by additional libraries and code that permit standard C to be used for designing parallel (multi-processor) applications. For example, the POSIX library provides support of multi-processor threads [4, 14]. An alternative to explicit thread-based parallelism is to create a compiler with the specific purpose of squeezing low-level parallelism from code generated with C software. This alternative is the heart of most C-to-RTL compilers.

C-to-RTL generators take sequential code and combine it with special constructs and extensions to identify and generate parallel processes. These tools allow control over the hardware design down to the wire and register level. The extensions that provide the full performance of the hardware are not runnable in the normal development environment since the typical microprocessor does not support functional parallelism [15]. The additional constructs could be wrapped in #ifdef statements that will make them invalid for desktop simulation.

Currently available C-to-HDL compilers, covering open source and commercially licensed, include: Nallatech's DIME-C [2], Impulse Accelerated Technology's Impulse C [4], National Semiconductor's NAPA C [16], Mitrionics' Mitrion-C, SRC's Carte, Mentor Graphics's Catapult-C, Celoxica's Handel-C, Los Alamos's Trident compiler [2, 3, 12, 17]. The C-to-HDL tools that target FPGAs do not conform to ANSI C standards. Each of these tools uses a version of the C that differs from the ANSI standard, some vary considerably more than others. DIME-C, for example, represents a subset of C, while Impulse C and Handle C are supersets of C with proprietary additions. Others, like Mitrion-C, diverge significantly from the typical C programming format.

Regardless of the compiler used, it is never as simple as creating a typical ANSI C program. In order to obtain optimal performance some thought must be given to specific algorithmic development techniques that support parallelism [2, 4]. A C-to-RTL compiler generates an intermediate code (HDL) that will then pass through a second compilation that produces the synthesis files to be mapped and routed onto the FPGA logic fabric. Both the initial and second compilation processes have been automated over the years.

FPGA-based solutions can exploit an application's inherent parallelism; with the added benefit that hardware acceleration mechanisms, such as pipelines and special purpose computational hardware, can be generated for increased throughput. Therefore, designing an algorithm for maximum parallelism should lead to a near-optimal outcome since performance is greatly impacted by available resources and the parallelizability of the application.

Even though most compilers do some kind of automatic optimization, the programmer needs to provide adequate structures to maximize parallelism. To achieve a hardware acceleration of significant magnitude certain C programming techniques need to be applied that take into consideration the fundamental differences between a microprocessor and a FPGA [2, 18]. For example, if multiple accesses to memory, for reading/writing data from/to a block array, are performed in the original C code, then constraining the number of accesses to memory can significantly improve speedup because only one value can be read or written to memory in a single cycle. Alternatively, the array can be partitioned into multiple smaller arrays which will be implemented as separate physical memory structures within the FPGA, allowing parallel memory accesses. Also depending on the algorithm and the resources available a loop might prove more efficient than repeated macros [18]. In hardware, calling macros multiple times corresponds to duplicating the macro resources numerous times, which greatly increases the risk of exhausting limited logic resources.

Impulse C Programming Environment

Impulse C is a C-to-HDL compiler from Impulse Accelerated Technology (IAT), which was founded in 2002. The origins of this compiler can be traced back to the development of the Streams-C compiler research project at Los Alamos National Laboratories. However, a major difference between IAT's compiler and Stream-C is its compliance with ANSI C standards. Impulse C code can be written and debugged in any ANSI standard C environment, including Microsoft Visual Studio^{TM} and GCC-based tools. This is consistent with Impulse-C's aim to bridge the hardware domain gap for software programmers. The Impulse C programming language is part of the CoDeveloper tools that facilitate the creation of mixed software/hardware programs. The Impulse C IDE comes with tutorials that help the developer explore the compiler features and support levels available for various hardware targets.

Impulse C can be used for FPGA-accelerated computing with an embedded or external CPU host or to simply generate HDL modules. In our case, the interest is on hardware acceleration for an embedded processor. Impulse C contains API function libraries for parallel programming that fit into the standard C syntax. It can be used for system-level parallelism by taking advantage of the CSP programming model that underlies its parallelism identification capabilities. This tool provides the easy development of independent processes that can be interconnected to exploit the parallel execution of code as independent hardware blocks [14]. Applications and research areas where Impulse C have been used by software programmers include: lithographic aerial image simulation [19], boxcar filter and matrix-vector product [20], high-performance radix-2 FFT [21], image processing, digital signal processing, encryption algorithms, financial computing, and other arenas [4].

The essentials of an Impulse C program are processes and streams. Processes represent synchronized and independent sections of code that run concurrently, and streams provide the communication pathways between processes. These streams are executed as dual-clock FIFO interfaces in hardware and allow the parallelizability of code without having to resort to lower-level abstraction, or cycle-by-cycle synchronization. Impulse C also offers alternative programming modes such as shared memories and signals [4, 14]. The benefits of one mode over another are application specific. For example, programs that require multiple transmissions of fixed-size data packets, minimum signal synchronization, and no inter-processing dependencies are ideal for stream-based data exchanges between processes. Streams are always the choice when the data are being consumed as fast as it is produced or sent in a "sequential" format between hardware modules. However, shared memory is preferred for arbitrary accesses of data from one hardware process to the other [4].

Figure 5 shows Impulse C's design flow chart, indicating the steps from initial design to final implementation of an algorithm. For Impulse C and similarly for other C-to-HDL compilers, it is necessary to write and structure the program with parallelism in mind to obtain optimal performance. Once the design phase is completed, the stages of partition and simulation follow. For the partition phase, the source code is usually divided into a software component and a hardware component, where the software is in charge of input and output interfacing and the hardware portion is designated for the complex calculations. The software and hardware components consist of one or more processes that execute concurrently. In the stage labeled (desktop) simulation, the source code is debugged using standard C IDEs (Visual Studio^{TM}, CodeWarrior^{TM}, GCC, etc,) or with the Impulse C IDE. Upon finishing partition and simulation, the C-to-HDL compiler is called to automatically generate optimized VHDL code corresponding to the HW process and the appropriate interfaces for both the software and hardware components. The developer then uses platform-specific tools to synthesize the VHDL code. After synthesis, constraints and logic are mapped onto the target resources and a configuration bit stream is generated. In our case, the Studio Design Kit (EDK) from Xilinx generates the netlist and bit stream file to reconfigure the FPGA on the XUP Virtex II pro board. The user needs to specify certain synthesis tool configuration parameters such as bus sizes, port widths, and memory addresses that are applicable to the target platform.

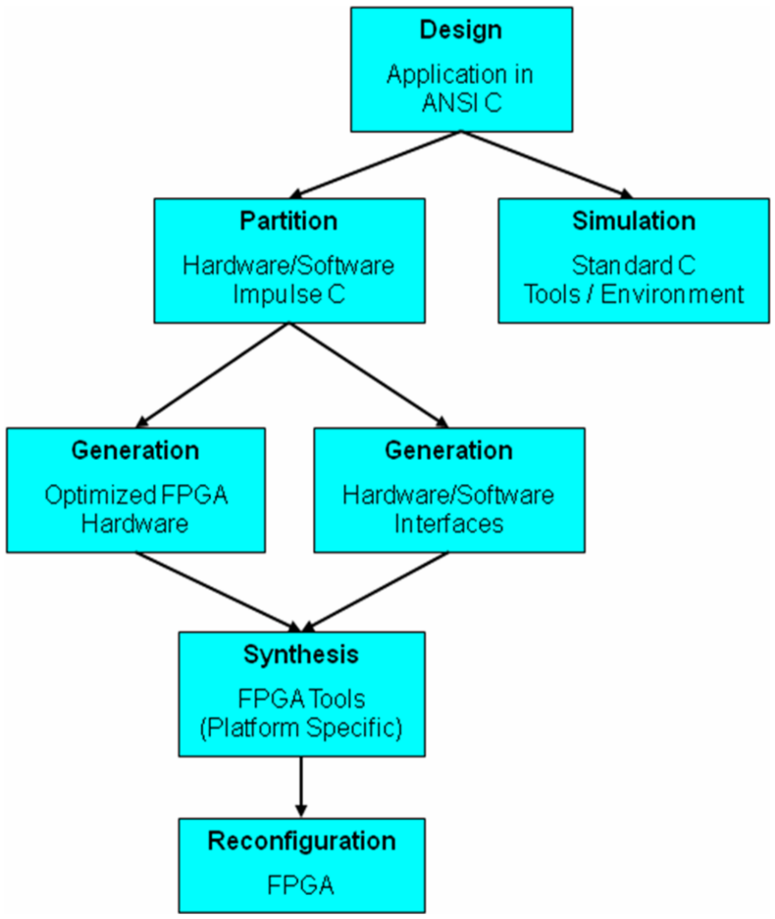


Figure 5: Impulse C Design Flow Chart

This C-to-HDL compiler provides a unified language to describe both software and hardware, which speeds up the development and verification of intellectual property or IP blocks and creates a design flow that enables embedded software engineers to develop complicated hardware components. Impulse C also supports, with some restrictions, the integration of lower-level code through the use of the pragma CO IMPLEMENTATION. The attached hardware code must be for internal computation only, i.e. the generated hardware cannot access external ports defined in the top level module. Contrary to typical software processes that permit only a single memory data access per transaction, hardware allows multiple reads and writes to memory within one cycle. In Impulse C, this advantage is exploited by performing array splitting. For this, the Impulse C tools can create separate memory blocks, each connected for local reads and writes by local hardware. This is useful for simultaneous computing or parallel processing.

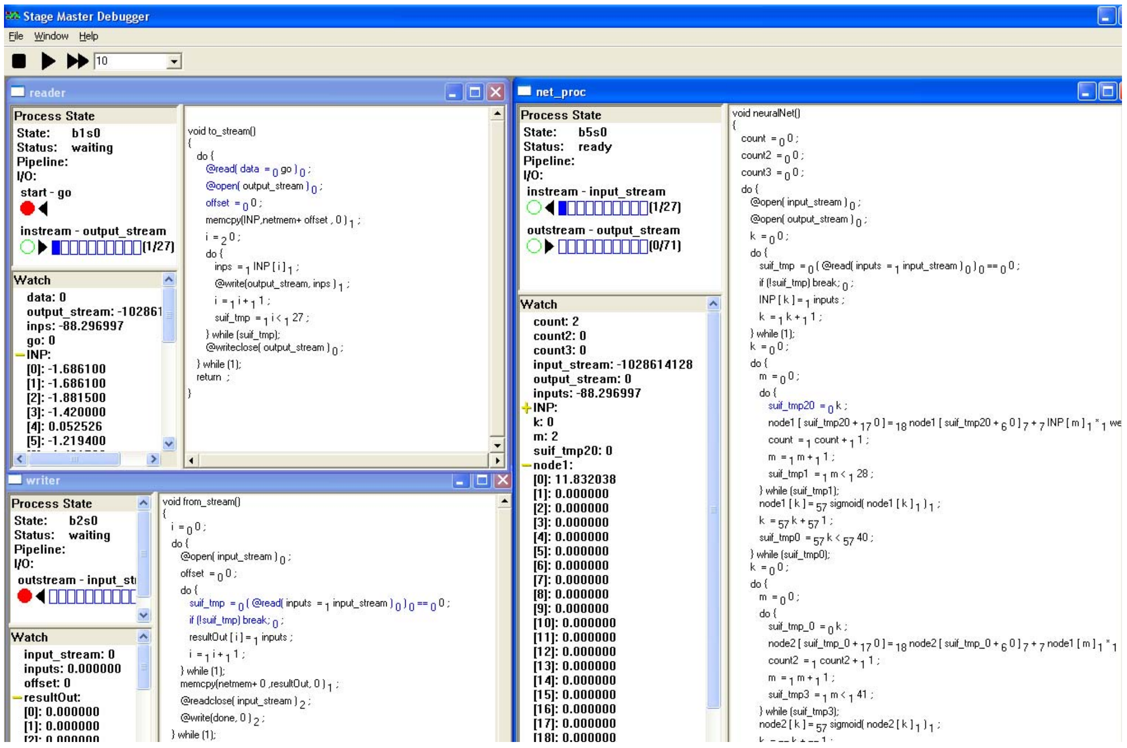
Impulse C also supports the IEEE standard floating-point data types that are appropriate for applications with a wide range of data magnitudes and incumbent accuracy requirements. Our neural network application benefits from this support because the inputs, weights and outputs are of continuous real data. With Impulse C, single and double precision floating-point data types are available by simply selecting a configuration *option* in the IDE *Project* menu. The Impulse C CoDeveloper tool creates the automatic link to the Xilinx CoreGen floating-point libraries necessary for the floating-point operations [22]. In the code itself the user simply needs to instantiate a *float* or a *double* type. However, it is important to make sure that the synthesis tool supports floating-point hardware inclusion. For example, only Xilinx EDK versions 9.1 and higher fully support floating-point inclusion.

The programming model based on streams fits very well to the development of algorithms in a FPGA-based platform that includes a dedicated coprocessor. In the Virtex II Pro, two hard core processors, PowerPCs, are embedded in the FPGA chip. The Impulse C compiler extracts low-level parallelism from each of the application's identified parallel processes and then automatically creates the inter-process communication interfaces that are necessary for the development of a joint hardware/software system. All this processing is transparent to the programmer's viewpoint. However, Impulse C's automatic C-to-VHDL code generation has limitations that become visible when external hardware resources are interfaced with the FPGA. The CoDeveloper tool has the capability to manage the external resources, but not necessarily with optimal performance, such as access to the DDR SDRAM. Fast on-chip Block RAM (BRAM) is a limited resource. Many applications will not fit into the available BRAM. A typical solution is to use the BRAM for booting the PowerPC processor prior to executing the application program stored in DDR RAM. Usually the external DDR memory is connected to the PowerPC via the main bus, PLB, because that it provides fast memory access. Impulse C currently does not support PLB access to the DDR SDRAM. This external memory resource must be accessed via the slower OPB bus which connects to the main bus using a plb2opb bridge core.

In addition to the Impulse C compiler, CoDeveloper tools' include CoMonitor Application Monitor for graphic visualization of interaction between processes as well as the Stage Master Debugger (SMD) that provides hardware simulation in a cycle-by-cycle format. The Stage Master optimizer provides information about pipelines performance in terms of the number of instruction stages, pipeline latency (in clock cycles) and the clock rate (in clock cycles). Impulse C groups code into operational segments to form a stage that executes in a single clock cycle.

Cycle Accurate Hardware Simulation in Impulse C can be realized in three ways: first, by using the hardware generated code, to create a VHDL test bench simulation; second, use the Stage Master Debugger tool to view a cycle-accurate C-language HDL representation; and third, run the synthesis tool and perform netlist simulations [4].

The Stage Master Debugger allows the user to step through each stage or cycle as in any common debugging environment, with the difference being that the designer can not step into a stage to process a single instruction separately. Figure 6 shows the main display of the Stage Master Debugger tool.



```
Process State
State:    b1s0
Status:   waiting
Pipeline:
I/O:
 start - go
 instream - output_stream (1/27)

Watch
  data: 0
  output_stream: -102861
  inps: -88.296997
  go: 0
 -INP:
  [0]: -1.686100
  [1]: -1.686100
  [2]: -1.881500
  [3]: -1.420000
  [4]: 0.052526
  [5]: -1.219400
```

```
void to_stream()
{
  do {
    @read( data =_0 go )_0 ;
    @open( output_stream )_0 ;
    offset =_0 0 ;
    memcpy(INP,netmem+ offset , 0 )_1 ;
    i =_2 0 ;
    do {
      inps =_1 INP [ i ]_1 ;
      @write(output_stream, inps )_1 ;
      i =_1 i +_1 1 ;
      suif_tmp =_1 i <_1 27 ;
    } while (suif_tmp);
    @writeclose( output_stream )_0 ;
  } while (1);
  return ;
}
```

```
Process State
State:    b2s0
Status:   waiting
Pipeline:
I/O:
 outstream - input_st

Watch
  input_stream: 0
  inputs: 0.000000
  offset: 0
 -resultOut:
  [0]: 0.000000
  [1]: 0.000000
  [2]: 0.000000
```

```
void from_stream()
{
  i =_0 0 ;
  do {
    @open( input_stream )_0 ;
    offset =_0 0 ;
    do {
      suif_tmp =_0 ( @read( inputs =_1 input_stream )_0 )_0 ==_0 0 ;
      if (!suif_tmp) break;_0 ;
      resultOut [ i ] =_1 inputs ;
      i =_1 i +_1 1 ;
    } while (1);
    memcpy(netmem+ 0 ,resultOut, 0 )_1 ;
    @readclose( input_stream )_2 ;
    @write(done, 0 )_2 ;
  } while (1);
```

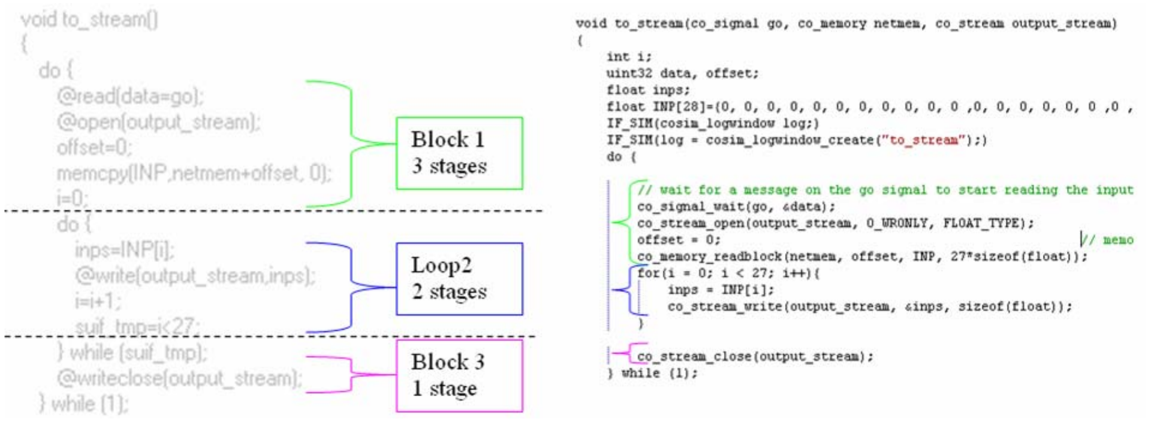
```
Process State
State:    b5s0
Status:   ready
Pipeline:
I/O:
 instream - input_stream (1/27)
 outstream - output_stream (0/71)

Watch
  count: 2
  count2: 0
  count3: 0
  input_stream: -1028614128
  output_stream: 0
  inputs: -88.296997
 +INP:
  k: 0
  m: 2
  suif_tmp20: 0
 -node1:
  [0]: 11.832038
  [1]: 0.000000
  [2]: 0.000000
  [3]: 0.000000
  [4]: 0.000000
  [5]: 0.000000
  [6]: 0.000000
  [7]: 0.000000
  [8]: 0.000000
  [9]: 0.000000
  [10]: 0.000000
  [11]: 0.000000
  [12]: 0.000000
  [13]: 0.000000
  [14]: 0.000000
  [15]: 0.000000
  [16]: 0.000000
  [17]: 0.000000
  [18]: 0.000000
```

```
void neuralNet()
{
  count =_0 0 ;
  count2 =_0 0 ;
  count3 =_0 0 ;
  do {
    @open( input_stream )_0 ;
    @open( output_stream )_0 ;
    k =_0 0 ;
    do {
      suif_tmp =_0 ( @read( inputs =_1 input_stream )_0 )_0 ==_0 0 ;
      if (!suif_tmp) break;_0 ;
      INP [ k ] =_1 inputs ;
      k =_1 k +_1 1 ;
    } while (1);
    k =_0 0 ;
    do {
      m =_0 0 ;
      do {
        suif_tmp20 =_0 k ;
        node1 [ suif_tmp20 +_17 0 ] =_18 node1 [ suif_tmp20 +_6 0 ]_7 +_7 INP [ m ]_1 *_1 we
        count =_1 count +_1 1 ;
        m =_1 m +_1 1 ;
        suif_tmp1 =_1 m <_1 28 ;
      } while (suif_tmp1);
      node1 [ k ] =_57 sigmoid( node1 [ k ]_1 )_1 ;
      k =_57 k +_57 1 ;
      suif_tmp0 =_57 k <_57 40 ;
    } while (suif_tmp0);
    k =_0 0 ;
    do {
      m =_0 0 ;
      do {
        suif_tmp_0 =_0 k ;
        node2 [ suif_tmp_0 +_17 0 ] =_18 node2 [ suif_tmp_0 +_6 0 ]_7 +_7 node1 [ m ]_1 *_1
        count2 =_1 count2 +_1 1 ;
        m =_1 m +_1 1 ;
        suif_tmp3 =_1 m <_1 41 ;
      } while (suif_tmp3);
      node2 [ k ] =_57 sigmoid( node2 [ k ]_1 )_1 ;
```

Figure 6: Cycle Accurate C Representation of HDL Code (3-process Neural Network)

The Stage Master Explorer (SME) generates a dataflow diagram that includes all hardware required elements. This tool also provides information about code segments and execution times in clock cycles. At the left side of Figure 7 is the C representation of the VHDL code generated by the compiler and at the right side is the C source code. Figure 8 shows a dataflow view of the code described in Figure 7. This graphical tool facilitates the analysis of each hardware process in the application by demonstrating the effectiveness of the compiler in parallelizing the original C source code.



```
void to_stream()
{
  do {
    @read(data=go);
    @open(output_stream);
    offset=0;
    memcpy(INP,netmem+offset, 0);
    i=0;
    do {
      inps=INP[i];
      @write(output_stream,inps);
      i=i+1;
      suif_tmp=i<27;
    } while (suif_tmp);
    @writeclose(output_stream);
  } while (1);
```

```
void to_stream(co_signal go, co_memory netmem, co_stream output_stream)
{
    int i;
    uint32 data, offset;
    float inps;
    float INP[28]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ,0, 0, 0, 0, 0, 0 ,0 ,
    IF_SIM(cosim_logwindow log;)
    IF_SIM(log = cosim_logwindow_create("to_stream");)
    do {

        // wait for a message on the go signal to start reading the input
        co_signal_wait(go, &data);
        co_stream_open(output_stream, O_WRONLY, FLOAT_TYPE);
        offset = 0;                                                    // memo
        co_memory_readblock(netmem, offset, INP, 27*sizeof(float));
        for(i = 0; i < 27; i++){
            inps = INP[i];
            co_stream_write(output_stream, &inps, sizeof(float));
        }

        co_stream_close(output_stream);
    } while (1);
```

Figure 7: C representation of HDL code and Impulse C Compared

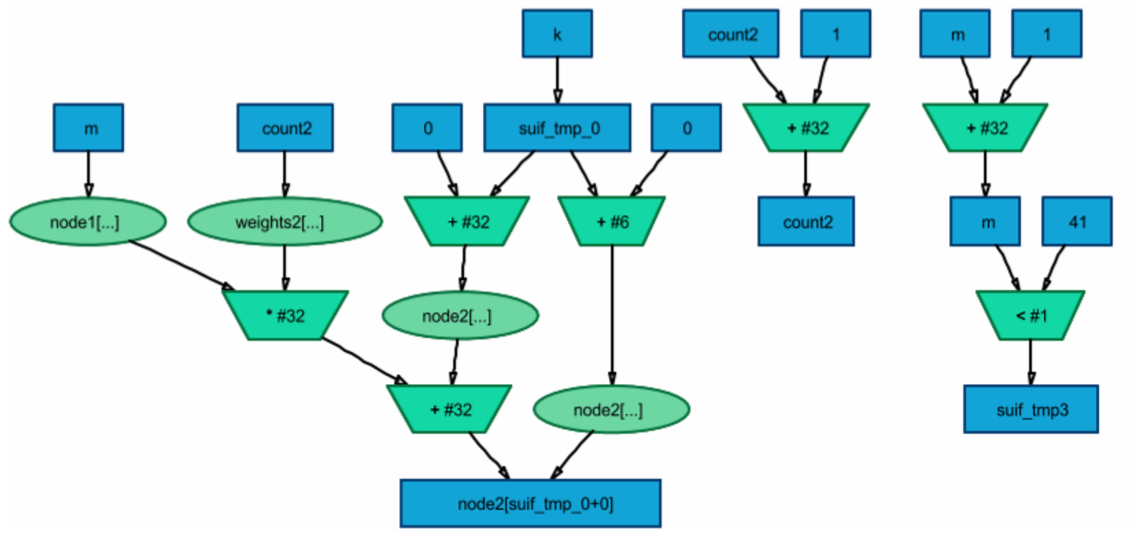


Figure 8: Stage Master Dataflow Graph for Neural Network Node 2

To obtain execution times for the neural network and compare with those obtained by the SMD tool, an OPB timer core, available for the XUP Virtex II Pro platform, can be used. The implementation of the timer is simple once the module is initialized. The timer can be reset before starting computations and then read via the *get_value* function at completion. The timer tick rate is linked to the OPB_Clk. Even though it might not be significant, calling the function to read the timer/counter also takes a small amount of time that will be included in the value returned from the counter. Therefore, it is important to first determine what value is obtained by resetting the timer and then immediately reading the timer. An accurate time can then be obtained by subtracting this time from the value read.

Comparison between C-to-HDL Compilers

Multiple C-to-HDL compilers have been developed, but all of them have a distinctive optimization technique. In the case of Handel-C, parallelism is achieved by explicitly sectioning the code and using extensions that tell the compiler to generate parallel hardware, while Dime-C and Trident do not require an open notification for code optimization but proceed to identify occasions for parallelism and pipelining automatically [2, 13]. Impulse C represents a “middle ground”, the compiler generates code with a certain degree of automated optimization, but also uses pragmas to explicitly convey pipelining and loop unrolling. There is no need to insert “par” or other RTL-equivalent statements to produce parallel logic. The CoDeveloper tool can extract some level of parallelism by grouping (staging) data independent code into blocks that can be executed simultaneously in a single clock cycle, keeping in mind that this can greatly increase the amount of hardware required as the block size increases in length.

In El-Araby, et al. [17] we see Impulse C compared to Mitrion C and DSP Logic in the Cray XD1 environment. In four different applications, the Impulse C implementation was not the most efficient, but the learning curve was not as steep as the other C to HDL tool, keeping Impulse C competitive for rapid prototyping. The cost for the Impulse C tools and hardware platform is much cheaper than the competition, some of which are as much as 15 times greater.

Another difference among C-to-HDL generators is their support for floating-point data types. Not all C-to-HDL tools accept floating-point data, although most provide some support. Handel-C and Impulse C can provide floating-point support based on the presence of specific hardware modules on the target platform, while other tools such as Trident, Dime-C and Carte C provide floating-point support based on libraries which implement specific floating-point operations independent of target hardware [2, 4].

CHAPTER FOUR

Neural Network

Concept, Structure, and Applications

A computation intensive application is chosen to investigate the efficiency of Impulse C and the FPGA's potential for reconfigurable computing and intrinsic parallelism. A trained artificial neural network is determined to be a good fit for the evaluation, especially since there are some prior results available to facilitate the comparison. Artificial Neural Networks (ANNs) are based on concepts derived from the structure and activity of biological neurons. Figure 9 shows an example of a simple neural network with 3 hidden layers.

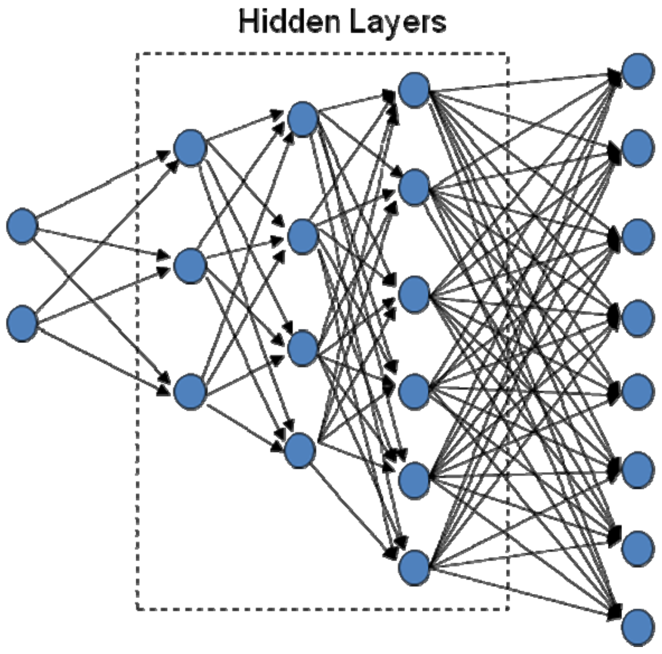


Figure 9: Neural Network Structure

The network can "recognize" patterns and relationships between a set inputs and parameters derived from training. Once trained with old data, the network can make a conjecture of the outcomes of new inputs. In order to predict the outputs, the inputs (i_0, i_1, i_2, … i_n) are multiplied by a set of weights which are then summed and passed through a non-linear 'squashing function' (f_s) at which point one node (o_j) of that hidden layer is formed as seen in Figure 10. This procedure is followed for all nodes of a particular layer. Once a layer's computation is finished, its nodes' outputs become the inputs for the calculations of the next layer. The process continues until reaching the layer prior to the outputs, in which the squashing step is omitted in the sonar application. The squashing function is a key part of the process since it accounts for non-linear relationships. For additional explanations on the nature and mathematical background see [23].

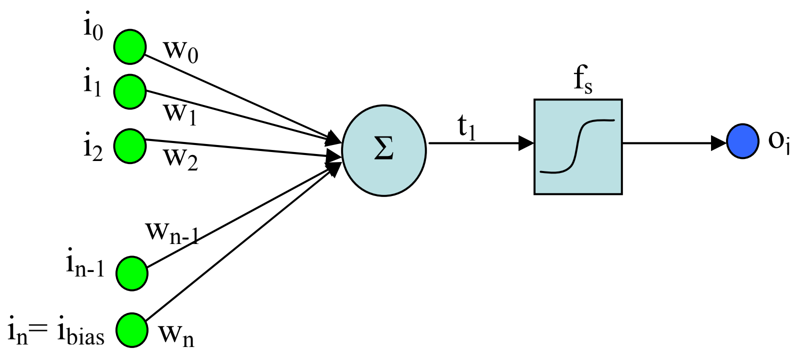


Figure 10: Single Node Computation

Some of the areas where neural networks are used include: pattern recognition in power systems [24], image processing, medical diagnosis [25], financial predictions, data mining, sequence recognition, gaming, system identification and control [26].

Previous Work Related to FPGA Implementation of ANNs

ANNs can be characterized computationally as parallel, modular and dynamically adaptive models. The calculations for each node in the same layer are independent; therefore, ideally all nodes of one layer can be computed in parallel. These characteristics make them good candidates for FPGA implementation. The only challenge being the limited hardware resources available in one platform to exploit the inherent parallelism in FPGAs for networks composed of a large number of neurons. Even though the multiplications (inputs times weights) that go into the formation of one node are independent, if the reprogrammable hardware does not contain enough resources (e.g. sufficient number of multipliers) to perform those computations simultaneously, maximum performance can not be achieved. Nevertheless, significant optimization is possible with FPGAs and as technology advances these devices should achieve even greater gate densities making them more suitable for such computationally intensive applications. A successful application of an ANN implemented in a FPGA is the hand tracking system of Krips, et al. [27]. Real-time performance was achieved by limiting the number of inputs, using 16-bits fixed-point data values, and generating weights (training coefficients) with a Matlab simulation [27],

The weights of ANNs traditionally have floating-point data precision, but the limitations imposed by the hardware resources have forced engineers to trade precision for logic area at the time of implementation. Nichols, et al. [28] studied the feasibility of floating-point arithmetic in FPGA based artificial neural networks using a single FPGA and concluded that such implementation is still not feasible and the 16-bit precision, which is considered the “minimum allowable precision” to maintain the learning capability of the network, provides the optimal solution. For this reason, ANNs are typically implemented in FPGA employing 16-bit fixed-point precision data representation.

Previous Work Related to FPGA Implementation of an ANN Trained to Emulated an Acoustic Model

To obtain an accurate comparison, our goal is to implement a neural network employed for acoustic undersea identification that was used previously in published research at Baylor University, but in this occasion employing Impulse C and XUP Virtex II Pro platforms. This network calculates the signal-to-noise ratio for a sonar system and consists of 27 inputs, 1,200 outputs and three inner (hidden) layers with 40, 50, and 70 nodes correspondingly. One additional node is added to each layer (input and hidden) to provide a bias term that improves the probability of discrimination flexibility. For example, the inputs for the second hidden layer (50 nodes) are the first hidden layer (40 nodes) plus one additional node, with the value of one, representing the bias term for a total of 41 nodes. All together the total number of multiplications reaches 91940 and 160 'squashing' or activation functions. It is safe to say that this problem represent a large amount of calculation requiring significant computational power.

This neural network application has been implemented previously in two hardware architecture as well as in two programming languages. The first implementation done by Burton Ottewell at Baylor University utilizes the SRC-6e platform and the Carte C programming language. The result was disappointing as the implementation ran slower than the equivalent software-only solutions running on a Pentium IV processor. Then, Paul Reynolds, as seen in [1], was able to create a significant speedup to the neural network computation using the same platform but using VHDL in a parallel implementation. To further explore the hardware implementation, Stephen Dark employed multiple XUP Virtex II Pro boards and VHDL to implement the same neural net. Through careful VHDL coding and combining three XUP board, further performance improvements were obtained by Dark. The aim of the present study is to evaluate Impulse C by implementing the same neural network and platform target as Dark and comparing the outcome to the three previous results from Burton, Reynolds and Dark. The results for the previous and current work will be discussed and summarized in a later chapter after the implementation and coding techniques are presented.

CHAPTER FIVE

Design and Implementation

Design and Verification Strategies

Until recently, the implementation of multiple independent floating-point operations in FPGA-based hardware was impractical because of the large number of gates required. But advances in logic density and the development of high-level development tools have facilitated the use of floating-point data types in FPGAs. In previous implementations of the target neural network application, fixed-point operations were utilized to reduce the number of required gates. However, to evaluate the applicability and the ease-of-use of Impulse C the weights were maintained and all calculations were performed in floating-point format rather than fixed-point. The fixed point implementation is the benchmark for comparison purposes. It is important to keep in mind that floating-point data representation will require more logic, but the objective is to determine the feasibility of the implementation given the availability of the resources in a Xilinx Virtex II device.

To investigate the feasibility of implementing a neural network application of twenty seven inputs and one thousand twelve hundred outputs, we started with a small network of two inputs and three outputs as seen in Figure 11. The first step was to divide the solution into two processes, a software process and a hardware process. The software process was implemented in the embedded PowerPC and is designated to send inputs to and receive the outputs from the neural network. The hardware process is in charge of the computations for the hidden layer nodes and the outputs of the neural network.

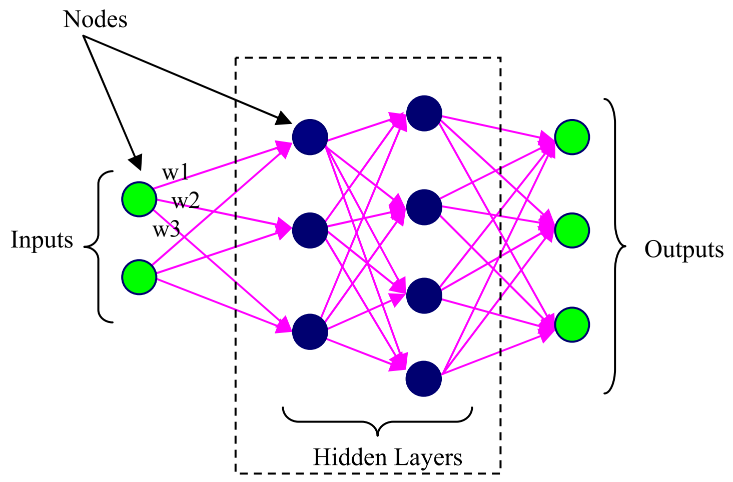


Figure 11: Small Neural Net

The hardware process was translated into VHDL code by the compiler. In this process, the inputs received from the software process via shared memory were multiplied by the appropriate set of weights. These products were then summed and passed through a squash function, which for our purpose is the logistic function defined in Equation 1 and as seen in Figure 12.

$$y(x) = \frac{1}{1+e^{-x}} \qquad \text{Equation 1}$$

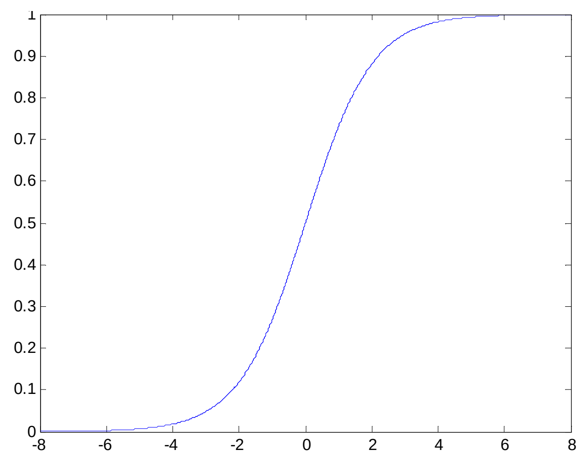


Figure 12: Logistic curve

Impulse C does not implement every transcendental function in the math.h library, e.g. the exponential function in Equation 1, but it does provide a pragma structure which allows the user to create functions such as power, factorial and absolute value. CoDeveloper tools allow you to use the exponential function as defined by the math.h library in desktop simulation mode, but during the generation of hardware an error message is generated indicating that it does not recognize the function. With this in mind, a Maclaurin series (Taylor series in the case that a = 0) expansion of the exponential function as described in Equation 2 was used.

$$e^x = \frac{x^1}{1!} + \frac{x^2}{2!} + ... \qquad \text{Equation 2}$$

Four functions, factorial(), power(), doNothing(), and expo() were created to implement the squashing process's exponential function. This method proved to be very inefficient, especially in the consumption of logic resources since these functions are used more than 150 times in the neural network computation. The creation of these four primitive functions represented a useful exercise to explore the *CO primitive* pragma. The hardware implementation of these functions resulted in extensive use of FPGA slices and in slow performance. The C code used by the hardware compiler for these functions is given below.

```
double power(float num, int pw)   // Power function
{
   #pragma CO primitive
   int counter      = 0;
   double results = 1;
   for(counter = 0 ; counter < pw ; counter++)
         results = results * num;
   return(results);
}

float factorial(float num)        // Factorial function
{
   #pragma CO primitive           //define function as a primitive
   int counter   = 0;
   float results = 1;
   float max    = num;
   for(counter = 0; counter < max ; counter++)
   {
         results = results * num;
         num--;
   }
   return(results);
}

float expo(float num)             //Exponential function
{
   #pragma CO primitive
   int counter   = 0;
   float results = 1;
   int max       = 140;
   for(counter = 1 ; counter < max ; counter++)
         results = doNothing(results)+power(num,counter)/
                   (factorial(counter));
   return(results);
}

float doNothing(float results)    //Do nothing function
{
   #pragma CO primitive
   return(results);
}
```

Alternative algorithmic methods such as Lookup Tables, Shift-add, CORDIC, and Taylor Segments Approximation (TSA) implementations, were explored by Paul Reynolds [1]. Reynolds found that the method with the best tradeoff balance between accuracy and logic and memory utilization was the Taylor Segments Approximation. This approximation was based on the second-order elements of the Taylor series expansion about specific points as indicated by equation 3 [1].

$$y(x) = -y_0''*(x - x_0)^2 + y_0'*(x - x_0) + y_0 \qquad \text{Equation 3}$$

The coefficients x_0, y_0, y_0', and y_0'' for the TSA with six segments are presented in Table 1. Greater accuracy can be obtained using additional terms.

Table 1: Taylor Series Segments and Coefficients [1]

Lower Bound	x_0	y_0''	y_0'	y_0
7.293	0.0	0.0	0.0	1.00000000000
4.771	6	0.001220703125	0.00244140625000	0.99755859375
3.317	4	0.008544921875	0.01757812500000	0.98205566406
2.482	2.75	0.045288085938	0.19653320312500	0.93994140625
0.425	1	0.045288085938	0.19653320312500	0.73107910156
0.0	0	--	0.25000000000000	0.50000000000

Once a functional model of the small neural network was implemented, the next step was to scale to the full-size neural network. The outputs were read and verified via a serial interface connected to an external HyperTerminal application and by reading from DDR memory via the XMD window. When developing the non-optimized neural network algorithm, the same double for-loop form was used to compute the hidden layers and the output nodes, with the omission of the squashing step for the outputs, as shown below.

```
// Compute nodes for the first hidden layer of the small neural net
for(k = 0 ; k < 3 ; k++)
{
   for(m = 0 ; m < 2 ; m++)
   {
         node1[k]+=INP[m]*weights1[count];
         count++;
   }
   node1[k]=1/(1+expo(-(node1[k])));         // squashing
}
```

The neural network calculations were verified during desktop simulation and the VHDL code was generated by the compiler. The first problem encountered when synthesizing the VHDL code was resource (logic gates) utilization. Even though the program was functional, it did not fit into the available hardware resources, thus causing an overmapping error. To address the lack of resources, the application was partitioned across multiple XUP boards, which was originally considered as an alternative for expanding the resources needed for parallelism. The computations for the neural network were distributed over three boards; one XUP board computed three hidden layers nodes and two hundred output nodes and the remaining two boards computed 500 output layer nodes each.

Communication and Partition between Boards

By taking advantage of the increasing logic density of FPGAs and advances in C-to-HDL compiler technology, it is our aim to raise the implementation efficiency and productivity of computationally intensive algorithms that require significant computing power. One consideration that must be addressed when using clustered FPGA-based platforms is the communication overhead created by cluster linkages. In the case of the C-to-HDL tools, most of these compilers facilitate design and prototyping when the algorithm fits into one board and the communication protocols are appropriate for single-board applications. The process of C to HDL translation in these cases is straight-forward and well-defined. The problem arises when the program surpasses the resources of one platform requiring additional external resources and interfacing.

Extensive computational requirements, floating-point data representation, non-standard communication modules, among other reasons, obligates the designer to deal with low-level hardware descriptions to meet the performance constraints of high performance computations applications. Our neural network application shares many of these characteristics. For example, it is implemented with floating-point data, and as expected, when implemented in the development platform, the neural net required more resources than was available in one XUP Virtex II Pro platform. In addition, the Aurora serial communication protocol was not readily supported by Impulse C, which required

HDL coding to generate and customize peripheral hardware to handle data transfers. The data between boards will be sent and received via SATA ports using the Aurora Communication Protocol as illustrated in Figures 13, 14 and 15.

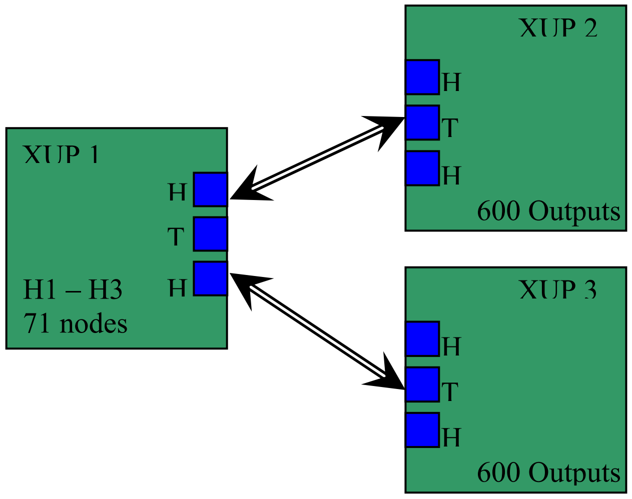


Figure 13: Three-Board Partition Version1

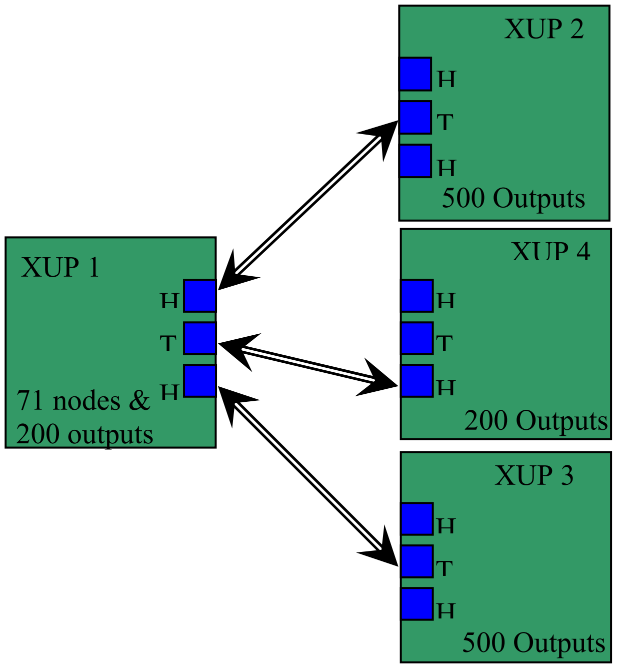


Figure 14: Four-Board Partition

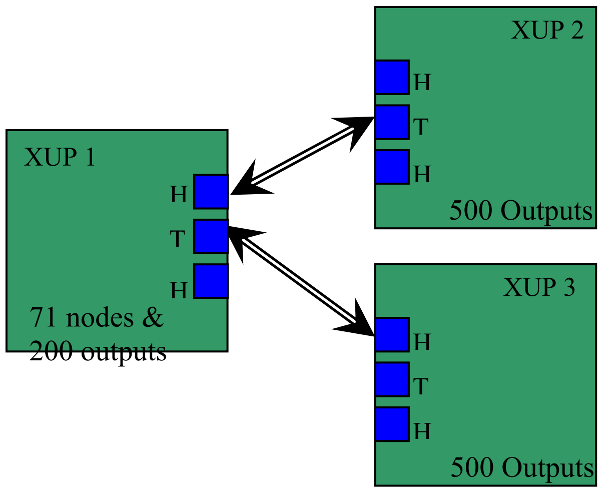


Figure 15: Three-Board Partition Version 2

Our present study used Xilinx IP cores and their corresponding XUP Virtex II Pro platform API drivers which can be readily combined with their Impulse C generated counterparts. The Aurora peripherals utilized in this study were modified versions of the single platform loop-back Aurora peripheral documented in [29]. The overall interaction between the platform boards and the Impulse Core can be seen in Figure 16. Each platform board contains a software process that distributes inputs to the hardware processes and an Aurora hardware IP core (light gray boxes) that provides communication between the software processes on each board.

Getting the first two boards communicating correctly represented a significant amount of work. Once a standard transmit/receive protocol was established, mirroring the process on the other boards consisted of creating an additional peripheral and initiating the communication process. Currently, serial-based communication processes between platforms is handled in software using a master/slave protocol, but it has the potential of expanded throughput using threads and multiple processors in a peer-to-peer protocol or by moving the protocol in its entirety to hardware. Communication processes are generally limited only by the hardware constraints of the platform (e.g. number of SATA ports or number of transceivers available).

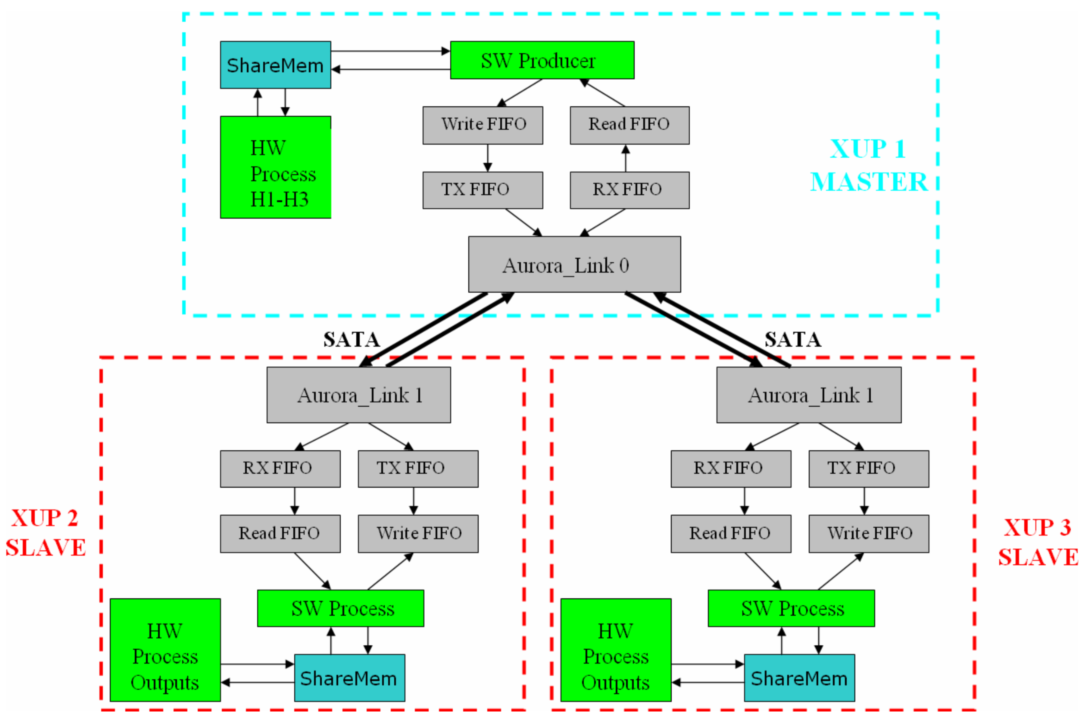


Figure 16: Neural Network Implementation using Impulse C and Aurora Communication

The communications model chosen for this study was a streaming interface; any data written into the transmitter buffer will be transported to the receiver buffer after some latency. A diagram of the AURORA streaming mode indicating the transmitting and receiving ports is illustrated in Figure 17. As soon as the communication channel is initialized, it is ready to send and receive data. When there is no data to send, idle characters are transmitted to keep channel open.

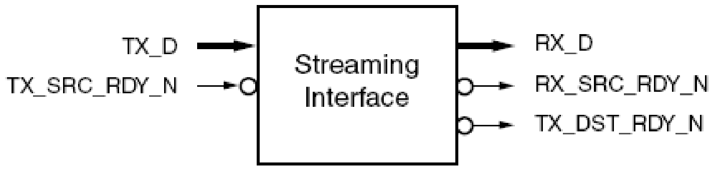


Figure 17: Aurora Core Streaming User Interface [18]

The a*urora_link* peripheral was generated by the EDK peripheral wizard, which creates a generic peripheral with the required wrapper for OPB or PLB bus interfaces and the corresponding drivers (written in C). HDL knowledge is necessary to customize the peripheral. The designer describes with VHDL or Verilog the behavior of the peripheral (e.g. defines how ports and signals of the different sub-components interact). The Xilinx Core Generator tool was used to create the a*urora_link* peripheral which includes two IP cores (modules), *aurora_stream* and *fifo_generator_v3_4096* plus user logic to establish the communication stream. Figure 18 shows the top-level structure of the peripheral with its specified input and output ports.

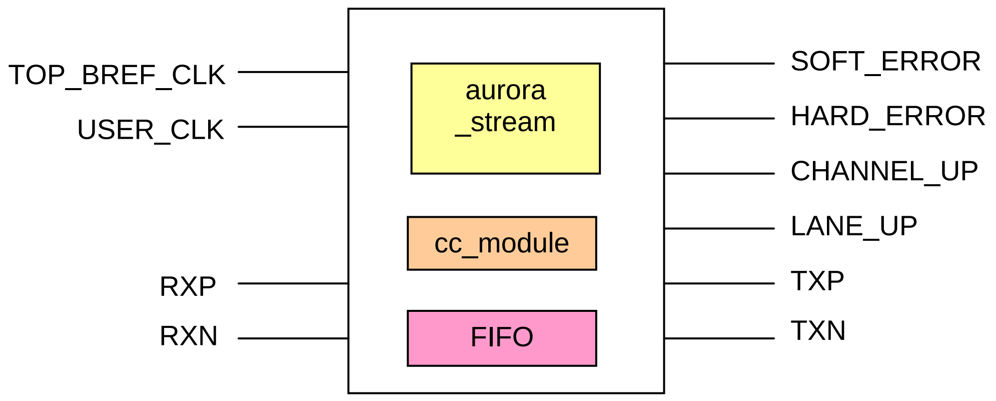


Figure 18: Aurora_Link Peripheral

As seen in Figure 16, the SATA communication system's structure can be characterized as a full-duplex (simultaneous bidirectional), master/slave channel. Since the channel's structure is rather simple with little explicit data control, a handshake protocol is needed to synchronize the exchange of data. The handshake needed to communicate from the master to the slave board is the same as the one used to connect the slave to the master platform; with the only exception being the acknowledgement messages. For the master-slave connection, the transmitter sends a thirty-two bit unsigned message signifying *ready-to-send, ((Xuint32)0x1234)* and waits for a thirty-two bit unsigned message indicating *ready-to-receive, ((Xuint32)0xABCD)* before writing data to the outgoing FIFO. The slave receiver, on the other hand, waits for a *ready-to-send* message before it sends the *ready-to-receive* acknowledgement and begins polling the receive FIFO for incoming data. In the case of slave-master connection, the *ready-to-send* message is *((Xuint32)0x9876)* and the *ready-to-receive* code is *((Xuint32)0xFEDC).*

An alternative method of synchronization uses hardware interrupts. Each aurora peripheral possesses interrupt capability, which can be used to indicate when it is ready to receive and when it is ready to transmit. All of these interrupts are handled by the OPB interrupt controller (OPB_INTR) core generated by EDK. The generated C code is added to the Impulse C code and combined by the linker-generator to register the program's exceptions and interrupts

Communicating via streams is easy and straight-forward except for the occasions when synchronization is necessary. Data arriving at the receiver can be over-written if not read immediately unless the receiver is equipped with a buffer (FIFO) to cache the arriving data. In our implementation, the algorithm sends 32-bit floating-point data across AURORA channels via SATA ports. Each Aurora peripheral has a FIFO attached to the receiver and transmitter. The optimal transfer size for the protocol is 16-bit packets, so 32-bit data values are broken into two halves for transmission and recombined as 32-bit value upon reception.

The handshake scheme is straight-forward, the slave board polls for data available at the receiving FIFO and check for a *ready-to-send* message. If such message is found, a *ready-to-receive* message is sent as an acknowledgment. After a connection is established, the transmitter will send the data accompanied with a *ready-to-send* code.

The receiver will check for the *ready-to-send* code that validates the data and begins accepting data as soon as it is available for as long a data is available in the receive FIFO. This handshake protocol fails when a *ready-to-send* signal is not received or lost, creating a block at the receiver. A similar problem occurs when both the transmitter and receiver functions are implemented with a loop that runs for a fixed number of iterations based on the number of values to be communicated. When an unexpected halt in the receiver occur, the result is non-terminating loop. Extracts of C code are shown below:

```
/****************************TRANSMITTING***************************/
  AURORA_MGT_mResetWriteFIFO(aurora_mgt_0_baseaddr);
  AURORA_MGT_mResetReadFIFO(aurora_mgt_0_baseaddr);
  xil_printf("XUP1 --> XUP2: Writing data: \r\n");
  do {
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdySend);
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdySend);

      while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
       {  }
      temp2 = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
    } while (temp2 != RdyRec);

  // Acknowledgement received, start sending data
  for(i = 0; i < 100; i++) {
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, testing);
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, (
         (short*)&floatArray[i])[0]);
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, (
         (short*)&floatArray[i])[1]);
  }

/**************************** RECEIVING ****************************/
  xil_printf("  Receive data in the AURORA_MGT_0 peripheral: \r\n");
  do{
     AURORA_MGT_mResetWriteFIFO(aurora_mgt_0_baseaddr);
     AURORA_MGT_mResetReadFIFO(aurora_mgt_0_baseaddr);
     i = 0;
     do{
         while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
           {   }
         temp = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
       } while (temp != RdySend);

     if (temp == RdySend)     //RdySend received, send acknowledgement
        AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdyRec);

     while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))      {  }
     if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
        receiving = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
     while (i < 100 ) {       //Read until reached # of expected inputs
        while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))   {  }
        if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
           receiving = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
        if(receiving == testing){
           if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
              ((short*)&floatArray[i])[0] =
                  AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
            if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
                ((short*)&floatArray[i])[1] =
                    AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
            i++;
        }
     }
  }
```

The difficulty with a polling mechanism is that an *end-of-stream* (*eos)* is required to indicate the end of the data sequence transmission and to ensure a complete transfer at the receiver. In the alternate implementation, an *eos* message was sent after the last valid data value. The result was that all available data was read, but not all the information was correct. Upon missing the *ready-to-send* message the data that follows is not read until there is a new *ready-to-send* flag. The *eos* method provided a way to check which sent data value was not received, but it does not stop the transmission; the net result being loss of data on the channel. Below is the code for transmitting and receiving with an *eos* message:

```
// handshaking messages
RdySend    = (Xuint32) 0x1234;
RdyRec     = (Xuint32) 0xABCD;
eos_sig    = (Xuint32) 0x3399;

/****************************TRANSMITTING****************************/
printf("XUP1 ->XUP2: Writing data:\r\n");
do {
      AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdySend);
      while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
          {  }
      temp2 = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
} while (temp2 != RdyRec);

// Once the RdyRec acknowledgment is received, send data
for(i = 0; i < 500; i++) {
     AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdySend);
     AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr,
          ((short*)&floatArray[i])[0]);
     AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr,
          ((short*)&floatArray[i])[1]);
}

/**************************RECEIVING***************************/
printf("  XUP2 <- XUP1: Reading data: \r\n");
do {
      while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
      {   }
      temp = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
} while (temp != RdySend);

// RdySend received, send acknowledgement, RdyRec
if (temp == RdySend)
    AURORA_MGT_mWriteToFIFO(aurora_mgt_0_baseaddr, RdyRec);

 while (index < 501 && (temp != eos_sig)) { //Read until eos received
    index++;

    // read accompanying data only if a RdySend is read first
    if(temp == RdySend){
       if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
          ((short*)&floatArray[i])[0] =
                AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
       if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr)) {};
          ((short*)&floatArray[i])[1] =
                AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
       i++;
    }
    else {
       // mark all the locations where a RdySend was not read
       missSpot[countermiss] = index;
       countermiss++;
    }

    // read FIFO, reading will be check against RdySend message
    while(AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
    {  }
    if(!AURORA_MGT_mReadFIFOEmpty(aurora_mgt_0_baseaddr))
       temp = AURORA_MGT_mReadFromFIFO(aurora_mgt_0_baseaddr);
}
```

The fundamental cause for the data loss can be attributed to delays and synchronization mismatches caused by automatic clock compensation. The Aurora protocol generates synchronization messages to prevent timing mismatches between the receiver and transmitter clocks. These clock compensation messages have the highest priority in the communication and can interrupt the flow of data. To verify the effect of clock compensation signals, a loop back test between transmit and receive ports on the same board was performed. The test showed that the data values could be lost, especially in the transmission of long sequences. A second test was performed with no clock compensation module, which resulted in a successful transmission and reception of data. Clock compensation is provided to ensure clock synchronization between boards at the time of transmission. Removing this module for the multi-board system is not a recommended option, because the clock rate of a particular board might vary enough to adversely affect communications between boards. For the large quantity of data being sent, for example 500 floating point values, it is reasonable to believe that receiver and transmitter clocks may vary significantly during the entire transmission duration. We were able to eliminate the clock compensation module for the *loop back* case for two reasons, first because we made sure that all data was being read until reaching an *eos* (*end-of-stream*) and second by counting the data received. An alternative is to modify the low-level clock compensation module to eliminate the conflict between the clock compensation messages and the data transfers.

The initial non-optimized serial communication protocol was very conservative and redundant. The slow communication rate resulting from the Aurora peripheral interface and the PowerPC through the OPB only added to its inefficiency. A *RdySend* message preceded each 32-bit value transmitted to ensure correct data recombination. This choice was safe, but increased the execution time of the application. Instead, a new strategy was implemented to mark the start of the transmission stream (*sos*), which would be sufficient to maintain the correct order for data recombination. Table 2 show the cycle clocks saved when switching from the *RdySend* transmission strategy to a single *sos* message option for communication synchronization.

Table 2: Timing Results Comparing Different Synchronization Schemes

Section	*RDY_SEND* Clock Cycles	*EOS_SOS* Clock Cycles	Difference Clock Cycles
200 Outputs received	461,215	461,346	-131
Transmit 71 nodes to XUP 1	532,896	512,171	20,725
Transmit 71 nodes to XUP 2	600,832	562,818	38,014
1 HS after Out ready	1,421,838	1,446,855	-25,017
500 outputs received	2,647,288	2,275,368	371,920
Next 500 outputs received	4,110,796	3,552,830	557,966

Below is the code for transmitting and receiving data between boards using a *sos*, *RdySend/RdyRec* handshake and *eos* messages.

```
/************** TRANSMITTING to SLAVE BOARD 70 NODES **************/
do {
   //Send Rdy to Send message and wait for responses
   AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, RdySend);
    while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
      {  }
      temp2 = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
} while ((temp2 != RdyRec) );
// Acknowledgement received, start sending data, send sos sig first
AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr,sos_sig);

for(index = 0; index < 71; index++) {
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr,
          ((short*)&tmp3[index])[0]);
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr,
          ((short*)&tmp3[index])[1]);
}
// Send end of stream (EOS)
AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, eos_sig);

/*************** RECEIVING from SLAVE BOARD 500 NODES **************/
do {
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
      {   }
      tempSlv = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
 } while (tempSlv != RdySendSlv);
// Acknowledgement received, start sending data
if (tempSlv == RdySendSlv) {
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, RdyRecSlv);
}
// start reading one SOS received
while (tempSlv != sos_sigSlv){
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
```

```
        {  }
        tempSlv = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
  }
  // Read FIFO until EOS
  while ((tempSlv != eos_sigSlv)) {
        while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
        {  }
        // Floating point data need to split into 2 half
        if(!AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
           tempSlv = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
        ((short*)&inputsSlv[iSlv])[0] = tempSlv;
        if(!AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
           tempSlv = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
       ((short*)&inputsSlv[iSlv])[1] = tempSlv;
       iSlv++;
  }
```

Neural Network Serial Implementation

Except for the partition of source code into hardware and software processes, the initial benchmark test was the execution time and logic resources used for the neural network forward computations with no optimization or parallel-like code enhancements. The first step was to write a program in ANSI C that computed each of the hidden layers and the corresponding outputs. For the example seen in Figure 16, the inputs A and B are employed to compute the first hidden layer nodes, C, D and E. In a serial implementation, one computation is performed at a time. First, input A is multiplied by a weight; second, the product is added to a temporary value, third input B is multiplied by a second weigh, and then that product is accumulated to the previous value. If there were other inputs, these would be multiplied by the associated weights and accumulated. At this point, the accumulated value is squashed to form the output at node C. This sequence of steps corresponds to the numerical order displayed in Figure 19. The same procedure is followed for the other nodes. The point of this illustration is to show that, serially, one computation is performed at a time and no parallelism is present. In C code these calculations are realized using a double for-loop.

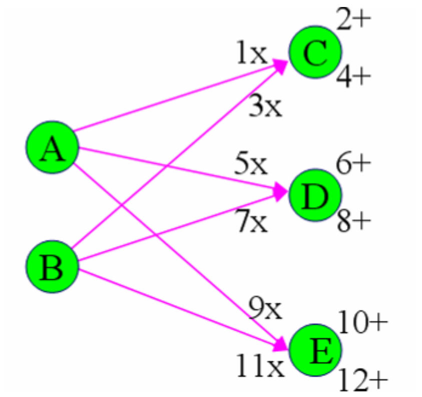


Figure 19: First Hidden Layer Computation – Serial Implementation

The inner loop generates the multiply and accumulate operations for each node (2 per node). The outer loop moves from node to node (C to D to E) and generates the squashed output for each node.

CO PIPELINE, CO UNROLL Pragma Directives for Parallelization

Pipelining is applicable when an algorithm within a loop executes in two or more cycles per iteration. For the above example, where a sum and a multiplication operation are required to compute the output a node, a pipelined implementation allows the two independent computations to be realized concurrently. After a pipeline latency of 2, the multiply and accumulate can calculated in the same clock cycle. Figure 20 shows the pipelined version completing the three node's calculations five steps earlier. To generate this pipelining behavior in Impulse C, it suffices to insert the CO PIPELINE pragma into the main loop of the algorithm. As simple as it sound, there are also restrictions and guidelines for optimal results. Pipelining applies to an inner loop and CO PIPELINE is prohibited for nested loops. It is also recommended that the pragma be inserted at the top of the code block instead of inside the code in which case it will require additional logic resource for parallelism, but will not result in significant optimization.

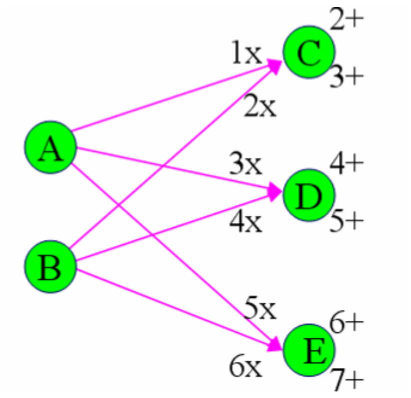


Figure 20: First Hidden Layer Computation – Pipelined Implementation

Parallelism can also be exploited at the loop-level by duplicating an operation over the span of loop iterations or what is known as hardware-based loop unrolling. The unrolled version of the small neural network example is seen in Figure 21. In this version the multiply and accumulate (MAC) resources were duplicated allowing parallel MAC computations. Instead of computing one product at a time, provided the availability of logic (gate) resources and non-data dependencies, multiple multiplications can be performed in one cycle. Similar to the CO PIPELINE pragma, Impulse C has a CO UNROLL pragma that automates the unfolding and hardware duplication of the targeted loop.

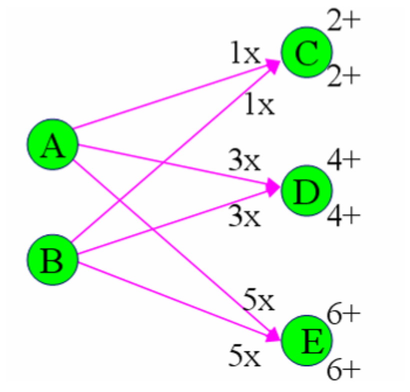


Figure 21: First Hidden Layer Computation – Unrolled Implementation

If pipelining and loop unrolling increases the performance of the algorithm, combining these two strategies creates even higher optimization. Figure 22 correspond to the small neural network implementation with both pipelining and loop unrolling. In this version after a pipeline latency of 1, multiple MACs can be executed simultaneously.

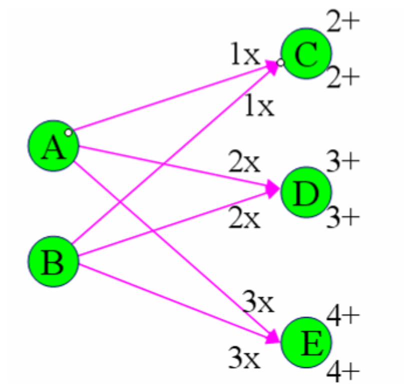


Figure 22: First Hidden Layer Computation – Pipelined & Unrolled Implementation

Pipelining and loop unrolling are two typical schemes to obtain parallelism. However, there are cases where the optimization is not so readily achieved or when it becomes contra-productive due to the excessive resources required or stalls in the pipeline flow.

A neural network application is a good candidate for investigating implementations with different levels of parallelism since these compute intensive applications are characterized by multiple non-data dependent calculations. Once the serial implementation was completed, the next step was to parallelize each of the double for-loops structures by inserting the CO PRIMITIVE pragma. Below is a source code fragment for the pipelined hidden layer and output computations (part 1) of the neural network.

```
// Pipeline each of the four loops that computes the NN nodes
#pragma CO NONRECURSIVE node1
```

```
#pragma CO NONRECURSIVE node2
#pragma CO NONRECURSIVE node3
#pragma CO NONRECURSIVE outputs

do{
   co_signal_wait(start,&res);
   co_memory_readblock(memblk,0,INP,27*sizeof(float));

   for(k = 0 ; k < 40 ; k++) {
         for(m = 0 ; m < 28 ; m++) {
         #pragma CO PIPELINE
               node1[k] += INP[m]*weights1[count];
               count++;
         }
         node1[k] = sigmoid(node1[k]);
   }

   for(k = 0 ; k < 50 ; k++) {
         for(m = 0 ; m < 41 ; m++) {
         #pragma CO PIPELINE
               node2[k] += node1[m]*weights2[count2];
               count2++;
         }
         node2[k] = sigmoid(node2[k]);
   }

   for(k = 0 ; k < 70 ; k++) {
         for(m = 0 ; m < 51 ; m++) {
         #pragma CO PIPELINE
               node3[k] += node2[m]*weights3[count3];
               count3++;
         }
         node3[k] = sigmoid(node3[k]);
   }


   for(k = 0 ; k < 200 ; k++) {
         for(m = 0 ; m < 71 ; m++) {
         #pragma CO PIPELINE
               outputs[k] += node3[m]*weights4[count4];
               count4++;
         }
   }
```

Ideally, the algorithm for the neural network computations would be realized by the code shown above; however, as mentioned earlier, some considerations are necessary when parallelizing an application. The parallelized algorithm was compiled and simulated (desktop simulation) and appeared to provide correct results. However, when implemented into the hardware platform, the results were not correct. When using the Stage Master Debuggers (SMD) for a cycle-accurate debugging, it showed erroneous results. The SMD provided additional information regarding to the source of errors. It showed that data was being read from multiple arrays in the wrong order. This tool was helpful in locating the problem. One solution for was to explicitly initialize all indexes before going into the pipelined loop, for example updating the *counter* variable in the outer loop. The approach worked for the SMD tool, but when implemented in the XUP board, the results were still incorrect. This highlighted the fact that hardware compiled in the cycle accurate debugger is sometimes different than the one synthesized and mapped onto the board. In EDK, although hardware routing was achieved, the complexity of the routing and the amount of logic resources required caused timing issues. To deal with these timing issues, the bus clock was reduced from 100 MHz to 50 MHz.

Different optimization schemes ("test cases") were explored as part of the iterative approach taken to evaluate Impulse C and implement the neural network application. In the first case, mentioned above, there was only a single hardware process and one primitive function that computed the hidden layer values and outputs. Figure 23 and Figure 24 show the first pipeline case with one hardware process.

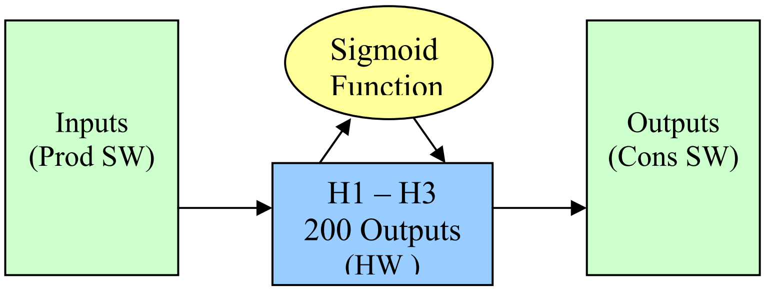


Figure 23: Case 1 – Two Software Processes and One Hardware Process

In case 1, the multiple array accesses in the computation of each node's value and the use the of high-latency option floating-point pipeline produced a result every 19 clock-cycles.

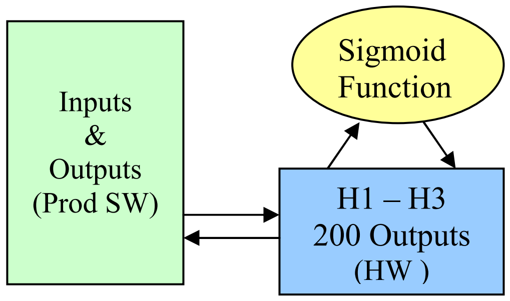


Figure 24: Case 1 – One Software Processes and One Hardware Process

Case 2 exploited Impulse C's CSP programming methodology. A different hardware process was created for each hidden layer, the sigmoid function and the output layer nodes as seen in Figure 25.

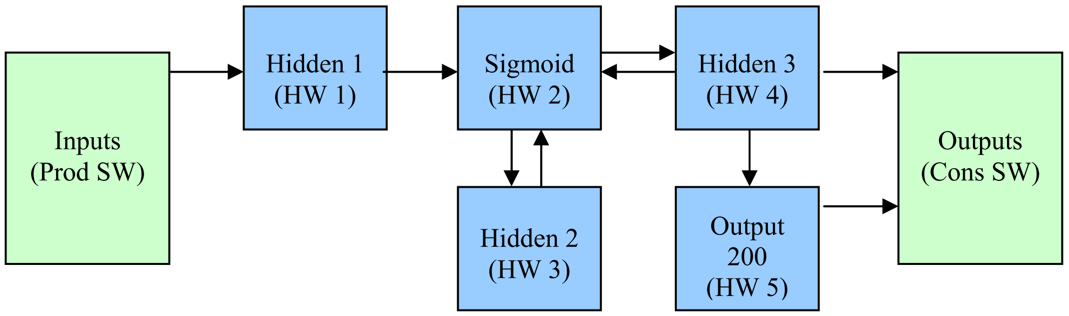


Figure 25: Case 2a – Two Software Processes and Four Hardware Processes

In case 2a the inter-process hardware communication is via Impulse C streams. Case 2a works in simulation, but co_streams are restricted to unidirectional point-to-point data flow between processes, which makes case 2a impossible to implement in hardware. A desktop software simulation can display proper behavior but fail to enforce the required hardware characteristics of Impulse C stream. To communicate across multiple processes, multiple streams are required in a point-to-point schema or additional hardware processes are needed as seen case 2b, see Figure 26 below.

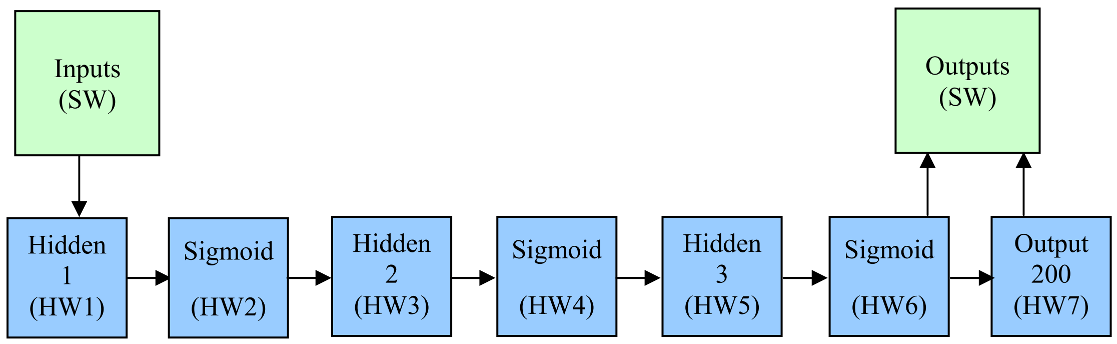


Figure 26: Case 2b – Two Software Processes and Seven Hardware Processes

Two main changes took place for case 2b: first, the low-latency option for floating-point implementation was set, which reduces the latency from nineteen to eight clock cycles, and the second, multiple concurrent hardware processes were used. These changes take advantage of the CSP structure that allows multiple hardware processes to execute at the same time. By employing these changes the total number of clock cycles in the pipeline application (CO PIPELINE pragma) was reduced to ~80,000 compared to ~170,000 required for the non-pipelined implementation on the master platform. In the pipelined version, reducing the output rate to half (8 cycles to 4 cycles) caused twice the speedup. It is important to note that if there are data accesses/updates of array elements in the pipeline, a slower clock must be utilized to meet timing requirements. For efficient usage of the pipeline pragma it is important to minimize the data access or pay the price by reducing clock frequency which diminishes the benefit of pipelining.

The CO UNROLL pragma further improves the design, but is constrained by the logic resources of the platform. Case 3 shows the use of the pragma in the implementation of the output layer values. Figure 27 and 28 illustrates loop unrolling process.

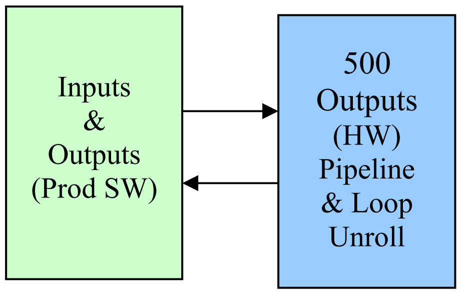


Figure 27: Case 3 – One Software Process and One Hardware Process Manually Unrolled

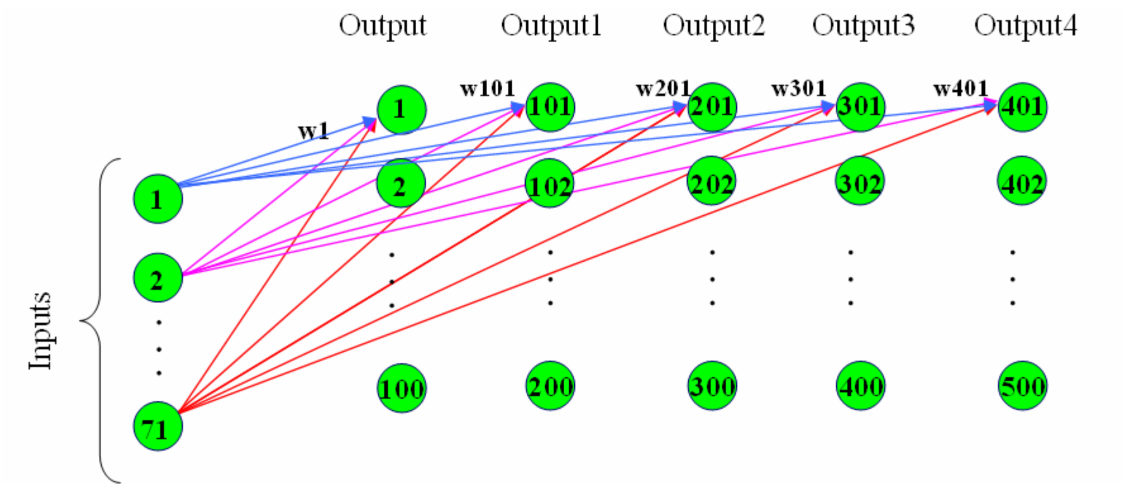


Figure 28: Case 3 – 500 Loop Unrolled into Five Nodes Computed at Once

The loop unrolling feasibility depends on the available resources of the target platform. The more operators that are duplicated, the more hardware resources are required for the implementation. Below is the C source code for the hardware process with the pipelining and loop unrolling implementation.

```
// Code to compute NN 500 outputs for slave board XUP2
// Exploring CO PIPELINE & CO UNROLL

#pragma CO NONRECURSIVE output2
#pragma CO NONRECURSIVE output3
#pragma CO NONRECURSIVE output4
#pragma CO NONRECURSIVE output4
#pragma CO NONRECURSIVE output

co_signal_wait(start,&res);

co_memory_readblock(memblk,35500*sizeof(float),INP,70*sizeof(float));

for(k = 0 ; k < 7100; k++)
```

```
{
#pragma CO PIPELINE
    test_inp = INP[m];
    wei4 = weights4[w];
    wei3 = weights3[w];
    wei  = weights[w];
    wei1 = weights1[w];
    wei2 = weights2[w];

    // Compute 5 output nodes in parallel
    test2 += test_inp*wei2;
    test3 += test_inp*wei3;
    test  += test_inp*wei;
    test1 += test_inp*wei1;
    test4 += test_inp*wei4;
    w++;
    m++;

    // One node formed
    if ( m > 70){
          output3[count] = test3;
          output4[count] = test4;
          output[count] = test;
          output1[count] = test1;
          output2[count] = test2;
          count++;
          m = 0;
          test3 = 0;
          test4 = 0;
          test = 0;
          test1 = 0;
          test2 = 0;
    }
 }
// write outputs to shared memory for SW process
co_memory_writeblock(memblk,400*sizeof(float),output4,100*sizeof(float));
co_memory_writeblock(memblk,0*sizeof(float),output,100*sizeof(float));
co_memory_writeblock(memblk,100*sizeof(float),output1,100*sizeof(float));
co_memory_writeblock(memblk,200*sizeof(float),output2,100*sizeof(float));
co_memory_writeblock(memblk,300*sizeof(float),output3,100*sizeof(float));
co_signal_post(done,count+count2+count3+count4);
```

CHAPTER SIX

Results

In the first serial implementation, with higher latencies and a fast clock option, the computation for each node in the first hidden layer requires 607 clock cycles. Each multiply and accumulate (MAC) stage takes nineteen clock cycles while a call to the sigmoid function costs seventy-four cycles. The computation for each node in the second hidden layer requires 854 clock cycles and 1044 clock cycles for each node in the third hidden layer. The output layer requires 1350 clock cycles. These values are greatly reduced when choosing the lower latency, slower clock implementation of floating-point operations. The latency is reduced to eight for each MAC operation in the main loop bringing the total time for one node for the first hidden layer to 269 clock cycles.

In the parallelized implementation with pipelining, the computation for each node in first hidden layer requires 112 clock cycles due to the rate of four in a pipeline with a latency of eight. The time required for a node to pass through the squashing step (latency 24, rate 1) can be absorbed in the following node's computation if a separate concurrent process is created. For the nodes in the output layers, the computations are the same as for the nodes of the other layers except for the squashing step, therefore, the timing is equivalent as well.

For the parallelized implementation with unrolling and pipeline implementation (feasible only for the output layer computations in our case 3) the timing per node computation is similar, but five nodes were computed simultaneously. However, in order to meet timing constraints the bus clock frequency was reduced 50 MHz to achieve the rate of four in the pipeline. If the 100 MHz frequency is desired, the pipeline rate must be eight, in which case the benefit of pipelining vanishes, but the benefit of loop unrolling is maintained. Table 3 summarizes the timing for each node in the various layers of the neural network. The table also includes the ideal timing value for the optimally parallelized case.

Table 3: Timing Comparison for One Node Computation

Neural Net Layer	Operation	Ideal (cycles)	Serial HL (cycles)	Serial LL (cycles)	Pipeline Rate 4 (cycles)	Pipeline& Unrolling (cycles)
Node in Layer 1	MAC	28	533	225	112	NA
	Sigmoid	1	74	44	1	
Node in Layer 2	MAC	41	780	329	164	
	Sigmoid	1	74	44	1	
Node in Layer 3	MAC	50	970	109	204	
	Sigmoid	1	74	44	1	
Node in Output Layer	MAC	71	1350	569	569 (100 Mhz) 284 (50 Mhz)	284/5 = 57

A rate of four was the best optimization possible using the current version of the Impulse C compiler because of how the algorithm is structured from basic components (e.g. summation and multiplication). The variable, *temp*, is and accumulator (e.g. it adds its previous value to the new product, which for the case of floating-point multiplication takes three cycles, and the conditional (e.g. if statement) in the main loop requires one cycle; thus resulting in a combined rate of four. In the Impulse C environment, a new execution stage is created for *switch-case*, *conditional* statements, and loop access (read or write) to memory. The conditional statement could be avoided if enough logic is available to compute one value per clock cycle (e.g. 71 floating-point multipliers for the worst case for a maximum of 71 inputs).

Table 4 shows the cumulative clock cycles parsed for the different stages of the neural network calculations. The timing results were obtained by reading the OPB timer register. These results points to the communication between boards as the main bottleneck. The Impulse C-implemented algorithm was executed in 64,207 and 34,113 cycles for the pipelined version with rate eight and four respectively. The average time to send 71 nodes was 54,561 cycles and the average time to send 500 nodes was 858,161 cycles. It took more time to send values from one board to another than to actually execute the algorithm. The overall timing for the neural network computation were 2,305,805 and 2,212,257 clock cycles for the pipelined implementations with rate eight and rate four. The timing improvements from one implementation to the other appear insignificant when comparing the overall application execution. Future improvements to the Aurora communications peripheral would better highlight the improvements achieved by a parallelized implementation.

In addition to the significant time required for communication between boards, the timing for the internal (same board) shared-memory communication between hardware and software processes is also very extensive for memory read and write operations. Currently external memory accesses to the DDR-SDRAM via the slower OPB processor bus create considerable overhead. In Table 4, the timing difference between values given in rows 13 through 17 shows the overhead associated with reading 100 values from shared memory: about 20,073 clock cycles.

Table 4: Cumulative Timing Results per Sections in Neural Network Computations

Row	EDK - OPB Timer	Pipeline Rate of 8 (clock cycles)	Pipeline Rate of 4 (clock cycles)
1	Impulse C Algorithm Completed (Done signal) 71 hidden layer nodes (outputs)	64,207	34,113
2	Receive 71 inputs	74,454	44,461
3	1st ack to receive (from SLV1)	77,188	47,138
4	Finished transmitting 71 values	131,276	108,699
5	1st ack to receive (from SLV1)	133,129	110,942
6	Finished transmitting 71 values	176,633	170,032
7	Impulse C Algorithm Completed (Done2 signal) 200 output layer nodes (outputs)	177,377	NA
8	Read 200 output from Mem	204,143	NA
9	XUP 2 receive 71 inputs	329,424	NA
10	write 71 inputs to share mem	341,526	NA
11	Send start signal from SW to HW (Impulse C)	393,239	NA
12	Impulse C Algorithm Completed (Done signal) 500 output layer nodes (outputs)	442,816	NA
13	Read 100 output from DDR	466,244	NA
14	Read 200 output from DDR	490,441	NA
15	Read 300 output from DDR	514,832	NA
16	Read 400 output from DDR	539,761	NA
17	Read 500 output from DDR	563,182	562,650
18	Receive 500 outputs from XUP 2	1,425,053	1,376,978
19	Receive 500 outputs from XUP 3	2,305,805	2,212,547

Floating-point data representation and operations are resource hungry, but provide a wider range of precision and facilitate the implementation of algorithm by avoiding the data type conversion to fixed-point. As mentioned previously, Impulse CoDeveloper automatically generates references to Xilinx COREGen floating-point libraries from C language statements and standard *float* and *double* data types. For low-latency floating-point implementation, the multiplication and addition operators are generated with latencies of 3. For high-latency floating-point implementations, the multiplication and addition operators, are generated with latencies of 6 and 11, respectively. Table 5 shows the XUP Virtex II Pro logic resources required for the implementation of single-precision (32-bit) floating-point multipliers and adders with various latencies.

Table 5: Logic Resources Used in the Implementation of Floating-Point Operators

Virtex-II PRO		FPGA Resources		FPGA Fabric		
Operator	Latency	Resource	Number	LUTs	FFs	Slices
Multiplier	8 (Max)	Logic(no usage)	0	621	687	424
Multiplier	4	Logic(no usage)	0	589	423	341
Multiplier	4	MULT18x18	4	177	211	160
Multiplier	3	Logic(no usage)	0	711	238	374
Multiplier	3	MULT18x18	4	166	159	129
Add/Subtractor	13 (Max)	Logic	0	560	576	458
Add/Subtractor	4	Logic	0	521	167	277
Add/Subtractor	3	Logic	0	511	140	270

Our neural network application is a low-latency floating-point implementation. Based on the *device utilization summary* produced by EDK, the logic implementation for the VHDL code (generated from C by the Impulse C compiler) that targeted the master board (hidden layer and 200 outputs) required 74 multipliers. From the Impulse C compiler tool, the seven hardware processes corresponding to the calculations for the master board used 17 floating-point multipliers (32-bit) and 29 floating-point adders (32-bit). See Appendix A for more details. This logic resource report implies that MULT18x18 logic blocks were used to implement the floating-point multiplications. Multiplying 17 times 4, results in 68 MULT18x18 devices, which added to 6 integer-based multipliers sums to 74. Other logic resources, including flip flops (FFs) and look-up tables (LUTs) are also employed.

The Virtex II FPGA resources are limited to 34 floating-point multipliers and adders if the device resources are not used for other purposes. For example, FFs and LUTs can also be used to implement memory if BRAMs are not available. Neural network applications which require large amounts of data storage, preferably in BRAM storage for fast access, face a limiting factor when data arrays are implemented using LUTs. The use of LUTs for storage reduces the resources need to implementing floating-point operators.

Table 6 summarizes the timing results for the neural network computation using different platform architectures and programming languages. The lowest run-time for a Impulse C generated neural network application was 22,125 μs. Comparisons with the timing results obtained from previous implementations were not encouraging. All but one (Pentium 4 floating-point implementation) were implemented using 16-bit fixed-point computations. However, factoring out the timing for communication, an execution time of approximately 800 μs is a closer match to the other results. It is also important to notice that the floating-point operators are implemented with a latency of three by the compiler, which could be reduced further to attain a higher throughput for the pipeline and a faster overall execution time.

Table 6: Timing Results for One Pass Forward Neural Network Calculations

Architecture	Language	Execution Time
PC – Pentium 4 (1.8 GHz)	C	250 μs
SRC-6E	SRC Carte^{TM} (parallel)	572.55 μs
SRC-6E	VHDL (serial)	1000 μs
SRC-6E	VHDL (parallel node)	250 μs
SRC-6E	VHDL (parallel input)	15 μs
XUP Virtex II Pro	VHDL (1 board)	15 μs
XUP Virtex II Pro	VHDL (3 boards)	6.7 μs
XUP Virtex II Pro (Float)	Impulse C (3 boards)	22,125 μs
XUP Virtex II Pro (Float -Ideal)	Impulse C (3 boards)	800 μs **
XUP Virtex II Pro (Int - Ideal)	Impulse C (1 board)	14.1 μs***

** 1200 floating-point nodes computed, time does not account for Aurora transmission
*** 1260 integer nodes computed without squashing function

For comparison-purposes, a 16-bit integer implementation with MAC operations only (no squashing processing) was implemented using Impulse C with a result of 14.1 µs. This result showed that fast implementation of computation intensive applications are feasible with Impulse C although limited by the target platform hardware resources as well as the application data-representation nature.

The floating-point implementations developed in this work showed limited usefulness primarily because of the limited XUP Virtex II Pro board resources. However, advances in FPGA technology promise higher gate densities in the future that will increase their usability for neural net applications or other applications that benefit from the larger dynamic range of floating-point computations. The fixed-point alternative was also explored in Impulse C by using macros defined in the compiler's math.h library. Even though the fixed-point implementation required fewer resources, the time spent to develop and simulate the results was considerable. The conversion from floating-point to fixed-point data and operations was not a trivial task, as factors such as integer bit-width, faction bit-width, overflow and saturation must be considered to obtain useful results. As an example, a code fragment for the floating-point version of the squashing function required one line of code while the fixed-point version required seven lines of codes, multiple macros and temporary variables and a more in-depth verification for correct results. Below are the code fragments for each version, floating-point and fix-point, respectively.

```
// Floating-Point Implementation
Node = -y000*((Node-x0)*( Node-x0)) + y00*(Node-x0) + y0;

// Fixed-Point Implementation
subtemp   = FXSUB16(Node,x0,5);
mult1temp = FXMUL16(subtemp,subtemp,5);
mult2temp = FXMULTS16(y000,mult1temp,12,5,8);
mult3temp = FXMULTS16(y00,subtemp,12,5,8);
mult4temp = FXMUL16(mult2temp,isnegval2,8);
```

```
    addtemp   = FXADD16(mult3temp,y0,8);
    addtemp2  = FXADD16(addtemp,mult4temp,8);
```

In the fixed-point implementation, the fixed-width bit field is divided into three parts, sign, integer and fraction. Typically, the sign bit is the most significant bit (MSB), followed by the number of bits that represent the integer part of the floating-point number and the last portion correspond to the fractional bits. Because some numbers require greater fraction accuracy while others require greater magnitude multiple16-bit fixed-point formats must be used. The weights had the form 1s7.8, which correspond to one sign bit, seven bits for the integer part and eight for the fractional portion. The format for the hidden layers computations (products and sums) was 1s10.5 while for the output layer computations were 1s7.8. For the sigmoid process, each segment of the TSA had the format of 1s3.12. More details of the implementation are shown in Appendix C.

CHAPTER SEVEN

Conclusions and Final Recommendations

Impulse C facilitates the exploration of partitioning an algorithm across multiple FPGA-based platforms. By automatically generating the VHDL code and allowing mixed SW-HW codesign and debugging, rapid-prototype-based designs can be quickly developed and compared. Even though the current floating-point implementation results were not desirable, increasing the number of platforms from three to fourteen such that all 71 floating-point computations could be performed in parallel would reduce the overall computation time to approximately 60,000 clocks (4 cycle per FP operation times 1500 computations) to provide greatly improved performance. Because each board is limited to five floating-point multipliers, fourteen boards would be required for a 71 multiplier implementation. Even with a cluster of fourteen boards, the cost of the reconfigurable computing machine would be well under $50,000. However, the communication overheard between boards would need to be significantly reduced.

Impulse C reduced considerably the prototype development time for the neural network application, but currently it does not include board-to-board stream libraries for communication at the hardware level (e.g. via parallel or serial interfaces). By extending the existing stream libraries with support for board-to-board communication at the hardware level, Impulse C would be a much more valuable tool for a broader range of targets and applications because the communication time could be made negligible. This type of support is provided by the SRC-6e/Carte C environment.

Even though EDK provides many wizards for peripherals and IP core utilization, hardware knowledge is required for the customization of most FPGA generic peripherals. Understanding of the functionality and capabilities of the programming environment and target platform had a significant impact on each of the design and implementations in this study.

Impulse C's iterative approach to optimization and its graphics tools provided a convenient way to explore and exploit parallelism. This C-to-HDL compiler enhances design productivity by providing fast HDL code development and easy implementation of common applications on widely used standard platforms. However, the compilers use of pipelining can be considered a high-level system optimization rather than a low-level one. Even though pipelining is available, many floating-point pipeline operations execute in more that one clock cycle. One potential improvement to the multi-cycle latency of floating-point operations is for Impulse C to allow more control to the designer to specify a numeric value latency (e.g. 1, 3, 4, etc) instead of the pre-defined low- and high- latency settings.

For VHDL code generation, a couple of hundreds of lines of C code generated multiple pages of VHDL code. The VHDL code generated is readable and maintains most of the variable names, so it is easy to trace back to the original C source code. Overall the Impulse C programming environment has the potential to fill the conceptual gap between software developers and hardware codesign prototyping.

The Aurora communication protocol represented the main bottleneck in the distributed neural network implementation. For future optimization a direct Impulse C – AURORA compatible interface should be designed and implemented. This would require a more thorough understanding of the hardware platform and the XML language needed to define the platform’s components in a manner such that the Impulse C compiler correctly generate the needed code.   A direct hardware communication would more than likely speedup the communication between platforms.

APPENDICES

APPENDIX A

Logic Resources

Table A.1: Logic Utilization for Sigmoid Function/Process

	Sigmoid 1 (multiple functions)	Sigmoid 2 (TSA as primitive function)	Sigmoid 3 (TSA as hardware process)
Total Stages	68	90	27
Operators	2 comparators (32 bit)	1 comparator (2 bit)	2 comparator (2 bit)
	3 adder/sub (32 bit)	8 comparators (32 bit)	6 comparators (32 bit)
	1 FP Adders/Sub (32 bit)	3 adder/sub (32 bit)	1 adder/sub (32 bit)
	1 FP Multiplier (32 bit)	8 FP Adders/Sub (32 bit)	8 FP Adders/Sub(32 bit)
	1 FP Adders/Sub (64 bit)	3 FP Multiplier (32 bit)	4 FP Multiplier(32 bit)
	1 FP Multiplier (64 bit)		
	1 FP Divider (64 bit)		

Table A.2: Logic Utilization for Hidden Layer Nodes Computations

	NN small	NN large (H1–H3) & 200 O
Total Stages:	105	267
Operators:	6 comparators (32 bit)	8 comparators (32 bit)
	14 Adder/sub(32 bit)	17 adder/sub(32 bit)
	3 Adder/sub(2 bit)	1 FP Adders/Sub(32 bit)
	7 FP Adders/Sub(64 bit)	4 FP Adders/Sub(64 bit)
	3 FP Multiplier(64 bit)	4 FP Multiplier(64 bit)
	2 FP Divider(64 bit)	2 Adder/Sub (6bit)
		1 Adder/Sub (7 bit)
		1 Adder/Sub (8 bit)

Table A.3: Logic Utilization for Output Layer Nodes Computations

	NN large 500 Out1	NN large 500 Out2
Total Stages:	267	267
Operators:	2 comparators (32 bit)	2 comparators (32 bit)
	5 adder/sub(32 bit)	5 adder/sub(32 bit)
	1 FP Adders/Sub(32 bit)	1 FP Adders/Sub(32 bit)
	1 FP Adders/Sub(64 bit)	1 FP Adders/Sub(64 bit)
	1 Adder/Sub (9 bit)	1 Adder/Sub (9 bit)

APPENDIX B

Software C Source Code for PowerPC

Slave XUP - Receiving

```
// Poll until a ready-to-send message is received from transmitter
do {
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr)) {  }
      temp = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
} while (temp != RdySend);

// Acknowledgement received, start sending data
if (temp == RdySend) {
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, RdyRec);
}
// Wait for the start of stream message before receiving data
while (temp != sos_sig) {
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr)) {  }
      temp = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
}
// Read from FIFO while the end of stream message has not been received
while ((temp != eos_sig)) {
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr)) {  }
      if(!AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
            temp = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
      ((short*)&inputs[i])[0] = temp;
      if(!AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
            temp = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
      ((short*)&inputs[i])[1] = temp;
      i++;
}
// Data shift scheme to balance the lack of clock compensation module
// If expected amount of data received, send it to shared memory for NN
if (i == 71) {
      co_memory_writeblock(memblk, 35500*sizeof(float), inputs,
                          70*sizeof(float));
}
else {
      // If not all data is received, signal an error message
      if(i < 71)
      {     printf("ERROR \r\n");   }
      // If additional data (0’s) is received, discard initial data
      else {
            for (index = 0; index < 70; index++)
            {     input[index] = inputs[i-(72-index)];  }
            co_memory_writeblock(memblk, 35500*sizeof(float), input,
                                70*sizeof(float));
      }
}
```

Slave XUP - Transmitting

```
// When ready to transmit send ready-to-send until acknowledge receive
do {
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, RdySendSlv);
      while(AURORA_LINK_mReadFIFOEmpty(aurora_link_0_baseaddr))
       {  }
      temp3 = AURORA_LINK_mReadFromFIFO(aurora_link_0_baseaddr);
} while ( (temp3 != RdyRecSlv));

// Send start of stream message (sos_sigSlv)
AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, sos_sigSlv);

// Send data
for(indexSlv = 0; indexSlv < 500; indexSlv++)
{
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr,
                              ((short*)&outpu[indexSlv])[0]);
      AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr,
                              ((short*)&outpu[indexSlv])[1]);
}

// Send end of stream (eos_sig_Slv)
AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, eos_sigSlv);
AURORA_LINK_mWriteToFIFO(aurora_link_0_baseaddr, eos_sigSlv);

/*********************** HANDSHAKING MESSAGES ********************/

RdySend    = (Xuint32) 0x1234;
RdyRec     = (Xuint32) 0xABCD;
RdySendSlv = (Xuint32) 0x9876;
RdyRecSlv  = (Xuint32) 0xFEDC;
eos_sig    = (Xuint32) 0x3399;
eos_sigSlv = (Xuint32) 0x88CC;
sos_sig    = (Xuint32) 0x259D;
sos_sigSlv = (Xuint32) 0x43AB;

/*********************** AURORA COMMUNICATION ********************/

// Outside of process, global pointer and variable
Xuint32 *aurora_link_0_baseaddr_p = (Xuint32 *)
XPAR_AURORA_LINK_0_BASEADDR;
Xuint32 aurora_link_0_baseaddr;

// Check that the Aurora peripherals exist
XASSERT_NONVOID(aurora_link_0_baseaddr_p != XNULL);
aurora_link_0_baseaddr = (Xuint32) aurora_link_0_baseaddr_p;

// Reset read and write FIFOs to initial state
AURORA_LINK_mResetWriteFIFO(aurora_link_0_baseaddr);
AURORA_LINK_mResetReadFIFO(aurora_link_0_baseaddr);
```

APPENDIX C

Hardware Process C Source Code for the Compiler to Generate VHDL code

Hidden Node Formation

```
void net_run(co_signal start, co_memory memblk, co_stream input_one,
co_stream output_one, co_signal done)
{
   // variable declarations omitted

   // weights
   co_int16 Inwei0[40]  = {   0xFD43,     0x0047,     0x0053,
      0x009A,     0x00BE,     0x0140,     0xFFA6,     0x0029,
      0x01DD,     0x0022,     0xFFEF,     0x002D,     0x01ED,
      0x0273,     0x0119,     0x03E7,     0xFF96,     0xFED8,
      0x000E,     0x0055,     0xFF53,     0x00B8,     0x007D,
      0xFFEF,     0x0007,     0x0049,     0xFFC6,     0x0026,
      0xFF3C,     0xFFD6,     0xFF8B,     0x000F,     0x0178,
      0xFF1A,     0xFED1,     0x0205,     0x0012,     0x0076,
      0xFE5F,     0xFDF0};
      .
      .
      .
   co_int16 Inwei27[40] = {   0x0DDF,     0xFD62,     0x0C49,
      0xFF67,     0xFDCB,     0xF167,     0x002B,     0xEEC0,
      0xFADF,     0x0032,     0xFF18,     0xF0CD,     0x05BB,
      0xF2E8,     0xF58E,     0xEB76,     0xFFA7,     0xEF26,
      0xFB3B,     0x00A6,     0xF1AC,     0xFD22,     0xF207,
      0x0087,     0xFFE7,     0xF36B,     0xF9B8,     0xFC95,
      0xFF77,     0x00C6,     0xF4F0,     0xF532,     0xF519,
      0xFC3A,     0xEF0B,     0xEFA3,     0xF3D0,     0x0107,
      0xFBEA,     0xF931};

   co_signal_wait(start,&res);

   co_memory_readblock(memblk,35500*sizeof(int16),INP,27*sizeof(int16))
;

   inp_rec = 0x0000;
   count   = 28;
   inp0  = INP[0];
      .
      .
      .
   inp27 = INP[27];

   sum0 = 0;
      .
      .
      .
   sum27 = 0;
```

```
inp_rec = 0;
counter = 0;

co_stream_open(output_one, O_WRONLY, INT_TYPE(16));

for (k = 0; k <40; k++) {
   wei0  = Inwei0[k];
       .
       .
       .
   wei27 = Inwei27[k];


   test0  = FXMULT16(inp0, wei0, 8, 5);
       .
       .
       .
   test27 = FXMULT16(inp27, wei27, 8, 5);

   sum0  = FXADD16(test0,test1,5);
       .
       .
       .
   sum13 = FXADD16(test26,test27,5);
   sum14 = FXADD16(sum0,sum1,5);
   sum15 = FXADD16(sum2,sum3,5);
   sum16 = FXADD16(sum4,sum5,5);
   sum17 = FXADD16(sum6,sum7,5);
   sum18 = FXADD16(sum8,sum9,5);
   sum19 = FXADD16(sum10,sum11,5);
   sum20 = FXADD16(sum12,sum13,5);
   sum21 = FXADD16(sum14,sum15,5);
   sum22 = FXADD16(sum16,sum17,5);
   sum23 = FXADD16(sum18,sum19,5);
   sum24 = FXADD16(sum20,sum21,5);
   sum25 = FXADD16(sum22,sum23,5);
   sum26 = FXADD16(sum24,sum25,5);

   // send unsquashed node to sigmoid process
   co_stream_write(output_one,&sum26,sizeof(int16));

   noSquash[counter] = sum26;    // nosquash value S10.5
   sum0 = 0;
       .
       .
       .
   sum27 = 0;

   test0 = 0;
       .
       .
       .
   test27 = 0;
   counter++;
}
co_stream_close(output_one);
```

Sigmoid (Hardware Process)

```
void sig_run(co_stream input_one, co_stream output_one)
{
    do{
      float f =0;
      int isneg  = 0;
      co_int16 x0   = 0;
      co_int16 y000 = 0;
      co_int16 y00  = 0;
      co_int16 y0   = 0;
      co_int16 tempNode = 0x0000;
      co_int16 numNode  = 0x0000;
      co_int16 subtemp  = 0x0000;
      co_int16 addtemp  = 0x0000;
      co_int16 addtemp2 = 0x0000;
      co_int16 mult1temp= 0x0000;
      co_int16 mult2temp= 0x0000;
      co_int16 mult3temp= 0x0000;
      co_int16 mult4temp= 0x0000;

      // Dummy variable represent -1 to temporary change negative
      // inputs to take advantage of odd property of sigmoid function
      int16 isnegval  = 0xFFE0; // -1.0 in fixed format S10.5
      int16 isnegval2 = 0xFF00; // -1.0 in fixed format S7.8
      int16 isnegval3 = 0x0100; //  1.0 in fixed format S7.8

      //x0 granularity S10.5 in order to match input for subtraction
      //y0, y00, y000 fixed point format S3.12 nature of coefficients

      co_stream_open(output_one, O_RDONLY, INT_TYPE(16));
      co_stream_open(input_one, O_WRONLY, INT_TYPE(16));

      while (co_stream_read(output_one, &tempNode, sizeof(int16)) ==
             co_err_none)
      {
            #pragma CO FLATTEN
            numNode = tempNode;
            if ( tempNode < 0) {
                  tempNode = FXMUL16(tempNode, isnegval, 5);
                  numNode = tempNode;
                  isneg = 1;
            }

            // segment ranges S10.5
            if(tempNode  < 0x006A) {
                  if(tempNode < 0x004F) {
                        if (tempNode < 0x000E) {
                              y000 = 0x0000; //0 in S3.12;
                              y00  = 0x0400; //0.25 in S3.12;
                              y0   = 0x0080; //0.5 in S7.8;
                              x0   = 0x0000; //0 in S10.5;
                        }
                        else {
                              x0   = 0x0020; //1 in S10.5;
                              y000 = 0x00BA; //0.045288085938 in S3.12;
                              y00  = 0x0325; //0.196533203125 in S3.12;
                        y0   = 0x00BB; //0.731079101563 in S7.8;
                  }
            }
            else {
                  x0   = 0x0059; //2.75 in S10.5;
                  y000 = 0x0066; //0.024780273438 in S3.12;
                  y00  = 0x00E7; //0.056396484375 in S3.12;
                  y0   = 0x00F0; //0.939941406250 in S7.8;
            }
      }

      else {
            if(tempNode < 0x00E9) {
                  if(tempNode < 0x0099) {
                        x0   = 0x0080; //4 in S10.5;
                        y000 = 0x0023; //0.008544921875 in S3.12;
                        y00  = 0x0048; //0.017578125000 in S3.12;
                        y0   = 0x00FB; //0.982055664063 in S7.8;
                  }
                  else {
                        x0   = 0x00C0; //6 in S10.5;
                        y000 = 0x0005; //0.001220703125 in S3.12;
                        y00  = 0x000A; //0.002441406250 in S3.12;
                        y0   = 0x00FF; //0.997558593750 in S7.8;
                  }
            }
            else {
                  x0   = 0x0000; //0 in S10.5;
                  y0   = 0x0100; //1 in S7.8;
                  y00  = 0x0000; //0 in S3.12;
                  y000 = 0x0000; //0 in S3.12;
            }
      }

      //sum26=-y000*((sum26-x0)*(sum26-x0))+y00*(sum26-x0)+y0;

      subtemp   = FXSUB16(numNode,x0,5);
      mult1temp = FXMUL16(subtemp,subtemp,5);
      mult2temp = FXMULTS16(y000,mult1temp,12,5,8);
      mult3temp = FXMULTS16(y00,subtemp,12,5,8);
      mult4temp = FXMUL16(mult2temp,isnegval2,8);
      addtemp   = FXADD16(mult3temp,y0,8);
      addtemp2  = FXADD16(addtemp,mult4temp,8);
      if (isneg)
      {
            addtemp2 = FXSUB16(isnegval3,addtemp2,8);
            isneg = 0;
      }
      co_stream_write(input_one,&addtemp2,sizeof(int16));
}
co_stream_close(output_one);
co_stream_close(input_one);
} while(1);
}
```

BIBLIOGRAPHY

[1] Reynolds, P. D. “Algorithm implementation in FPGAs demonstrated through neural network inversion on the SRC-6e,” M.S. Thesis, Dept. Eng., Baylor University, Waco, TX, May 2005.

[2] G. Genest, R. Chamberlain and R. Bruce, "Programming an FPGA-based Super Computer Using a C-to-VHDL Compiler: DIME-C," *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on,* pp. 280-286, 5-8 Aug. 2007.

[3] R. Bruce, M. Devlin and S. Marshall, "An elementary transcendental function core library for reconfigurable computing," in *Reconfigurable Systems Summer Institute (RSSI 2007),* 2007, pp. 1-9

[4] D. Pellerin and S. Thibault, *Practical FPGA Programming in C.* Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005.

[5] System C, http://www.systemc.org/home

[6] J. Hopf, G. S. Itzstein and D. Kearney, "Hardware Join Java: a high level language for reconfigurable hardware development," *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on,* pp. 344-347, Dec. 2002.

[7] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes and P. Chow, "A Scalable FPGA-based Multiprocessor," *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on,* pp. 111-120, April 2006.

[8] R. W. Brodersen, J. Wawrzynek, V. P. Srini, A. Vladimirescu, D. Orofino, J. Hwang, C. Chang, B. Richards, K. Camera, H. So, N. Zhou, "Reconfigurable HEC Platform", HECRTF Workshop, Washington DC, June 2003. [Online]. Available at HTTP: http://www.datafluxsystems.com/publications/hec_platform_hecrtf_work shop_ june_03.pdf

[9] Mitrion C, http://www.mitrionics.com/

[10] Xilinx, Inc., “LogiCORE Aurora User Guide”; UG061 (v2.7) , May 17, 2007.

[11] C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

[12] Verilog, http://www.verilog.com/

[13] J. L. Tripp, M. B. Gokhale and K. D. Peterson, "Trident: From High-Level Language to Hardware Circuitry," *Computer,* vol. 40, pp. 28-37, March 2007.

[14] D. Pellerin, "Streams-Based Programming Accelerates FPGA-Based Embedded Applications," *Embedded Magazine,* pp. 29-31, November 2006.

[15] J. P. Ardini, "Demand and penalty-based resource allocation for reconfigurable systems with runtime partitioning," in *Military and Aerospace Programmable Logic Devices. 8th. Held in Washington,* 2005, pp. 1-7.

[16] M. B. Gokhale and J. M. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture," *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on,* pp. 126-135, Apr. 1998.

[17] E. ElAraby, M. Taher, M. Abouellail, T. ElGhazawi and G. B. Newby, "Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study," *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on,* pp. 99-106, Feb. 2007.

[18] D. Pellerin and S. Thibault, “APP102: Optimizing Impulse C Code for Performance,” Dec. 2005, [Online]. Available at HTTP: http://www.impulsec.com/IATAPP102_OPT_TIPS.pdf

[19] J. Cong and Y. Zou, "Lithographic aerial image simulation with FPGA-based hardware acceleration," in *FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays,* 2008, pp. 67-76.

[20] P. Messmer, V. Ranjbar, D. WadeStein and P. Schoessow, "Advanced accelerator control and instrumentation modules based on FPGA," *Particle Accelerator Conference, 2007. PAC. IEEE,* pp. 506-508, 25-29 June 2007.

[21] Ardini, J.P. “A high-performance radix-2 FFT in ANSI-C for RTL generation,” Military and Aerospace Programmable Logic Devices. 8th. Held in Washington, DC, 09/07/2005 to 09/09/2005. Sponsored by: MAPLD. *(Draper Report no. P-4353)*

[22] Xilinx, http://www.xilinx.com/

[23] Reed, R.D., Marks, R.J. (1999) *Neural Smithing:* Supervised learning in feedforward ANN, Cambridge, MA: MIT Press.

[24] Bansilal, D. Thukaram and K. H. Kashyap, "Artificial neural network application to power system voltage stability improvement," *TENCON 2003. Conference on Convergent Technologies for Asia-Pacific Region,* vol. 1, pp. 53-57 o.1, Oct. 2003.

[25] M. Su and H. Chang., "Extracting Rules from Composite Neural Networks for Medical Diagnostic Problems," *Neural Process. Letters.,* vol. 8, pp. 253-263, 1998.

[26] Rome Laboratory, “Artificial Neural Networks Technology,” [Online document] Aug 1992, [Online]. Available at HTTP: https://www.dacs.dtic.mil/techs/neural/neural10.php

[27] M. Krips, T. Lammert and A. Kummert, "FPGA implementation of a neural network for a real-time hand tracking system," *Electronic Design, Test and Applications, 2002. Proceedings. the First IEEE International Workshop on,* pp. 313-317, 2002.

[28] Nichols, K., M. Moussa and S. Areibi, “Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks”, CAINE, San Diego, CA.

[29] Virtex-II Pro Resource, http://virtex2pro.blogspot.com/2008/03/create-aurora-transceiver.html.